



**Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza**

Proyecto Final de Carrera
Ingeniería en Informática
Curso 2012/2013

Visión por computador en dispositivos móviles para realidad aumentada

Alfonso Escriche Martínez

Septiembre de 2013

Directora: Ana Cristina Murillo Arnal

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

RESUMEN

Las técnicas de visión por computador se están extendiendo a los dispositivos móviles, gracias a la creciente mejoría de las especificaciones técnicas de los nuevos *smartphones*, que incrementan su capacidad computacional permitiendo aplicar algoritmos cada vez más complejos y costosos. Su aplicación estrella es la realidad aumentada, que se refiere a la combinación visual de elementos reales con virtuales.

Uno de los primeros ejemplos comerciales es *Wikitude*, que sale a la venta en 2008, y permite inicialmente insertar texto flotante superpuesto a la imagen capturada por la cámara, cuando el usuario se encuentra en unas coordenadas determinadas y apuntando la cámara en una dirección concreta. Otro paso lo dan empresas como *Layar* o *Qualcomm*, quienes introducen la realidad aumentada basada en marcadores. Sus productos son capaces de reconocer ciertas imágenes impresas en carteles, y superponer un modelo virtual sobre ellas. La última evolución llega al poder sustituir esos marcadores por imágenes reales. De esta forma es posible reconocer elementos del mundo real, y las aplicaciones pasan de depender de un marcador impreso a un elemento físico concreto. Sin embargo, solo se ha enmascarado la limitación del cartel impreso. Las aplicaciones siguen siendo dependientes de que el usuario apunte su cámara hacia algo concreto.

La mejora que este proyecto propone es detectar en tiempo real superficies planas en la escena, y proyectar sobre esta un modelo de realidad aumentada de una forma visualmente realista. Para ello se utilizan métodos de visión por computador que tratan de localizar este plano en la escena. Una vez determinada la superficie donde se puede mostrar la realidad aumentada, es necesario hacer un seguimiento de esta según se mueva la cámara. Es decir, si la cámara se mueve en una dirección, el modelo virtual debe también moverse acorde con el resto de la escena para mantener la sensación de realismo.

Dado que ya existen diversas soluciones comerciales que permite realizar este seguimiento y proyectar un modelo virtual, el principal objetivo de este proyecto es aplicar y comparar distintos métodos de visión por computador para realizar la detección de la superficie plana en tiempo real, y después inicializar el seguimiento aplicando alguna de estas soluciones comerciales.

Para demostrar esto se ha implementado un prototipo funcional para *Android* que ha sido probado en dispositivos actuales de gama media, un *tablet*, un *smartphone* y diversos dispositivos simulados o *ADV*, *Android Virtual Device*, con especificaciones técnicas propias de un dispositivo de gama media que el usuario medio posee. Se ha comprobado que es posible realizar lo que este proyecto propone, aunque debido a ciertas limitaciones de las soluciones comerciales de realidad aumentada, no se ha podido alcanzar un resultado listo para un producto comercial todavía.

Agradecimientos

A mi hermana Cristina, infatigable *betatester*, o más concretamente, infatigable acompañante en las calurosas mañanas de recolección de vídeos e imágenes como material de pruebas.

A mis amigos y socios Carlos y Mateo, pues fueron nuestras tardes de *brainstorming* las que me dieron las primeras ideas para hacer este proyecto; y es nuestro proyecto empresarial juntos el que me ha motivado a llevarlo a cabo con máxima ilusión.

También a mis amigos Ricardo, quien a veces parecía tener más interés en ver finalizado este proyecto que yo mismo, y Julio, a pesar de que aun estoy esperando el modelo tridimensional de la catedral de León que me prometió.

Por último, *aunque por ello más importante*, Ana Cris, guía más que directora de este proyecto. No solo por enseñarme y repetirme *n* veces hasta los más simples conceptos de visión, sino por reencaminarme las múltiples veces que perdía el norte.

Índice general

1. Introducción	1
1.1. Contexto y motivación	2
1.2. Objetivos y alcance	3
1.3. Metodología y entorno de trabajo	4
1.4. Trabajo previo relacionado	5
1.5. Organización de la memoria	6
2. Detección de superficies planas	9
2.1. Análisis general	10
2.2. Detección usando una secuencia	12
2.3. Detección usando dos frames distantes	13
2.4. Comparativa y pruebas	17
2.5. Conclusiones	21
3. Tracking y proyección	23
3.1. Análisis general	24
3.2. Integración de una solución comercial: Vuforia	26
3.3. Solución provisional: Optical Flow	29
3.4. Conclusiones	31
4. Prototipos	33
5. Conclusiones	37
5.1. Valoración personal	38
5.2. Trabajo futuro	38
A. API del framework desarrollado	43
B. Prototipo RA	47
C. Pruebas módulo 1	51
D. Pruebas módulo 2	65

Capítulo 1

Introducción

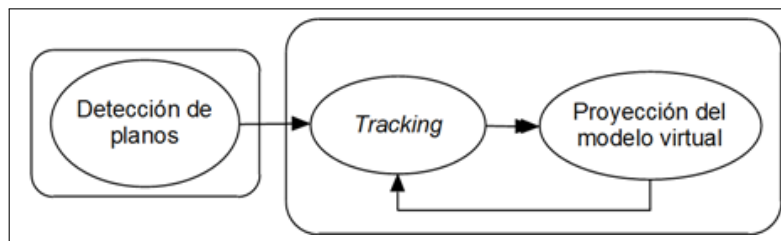


Figura 1.1: Diagrama general del proceso

Debido a la creciente capacidad computacional de los dispositivos móviles, están aumentando las opciones de utilizar técnicas de visión por computador para múltiples aplicaciones. La realidad aumentada es un ejemplo que proporciona nuevas funcionalidades y atractivos a las aplicaciones móviles. La visión por computador es la herramienta clave para incluir elementos artificiales en imágenes o vídeos capturados desde la cámara de estos dispositivos de manera realista. Sin embargo, muchos métodos actuales de realidad aumentada utilizan marcadores artificiales, cuyo funcionamiento se detalla más adelante, o no integran los elementos virtuales de acuerdo a la geometría de la escena, creando un resultado poco realista.

Este proyecto propone, por un lado, estudiar y evaluar las opciones de distintas técnicas de visión por computador (que analizan la geometría de la escena capturada por la cámara) para determinar dónde y cómo debe proyectarse un elemento gráfico de realidad aumentada. Por otro lado, implementar un prototipo realista de realidad aumentada que se pueda integrar en aplicaciones para móviles. El desarrollo del proyecto se ha dividido en dos módulos principales. El primero de ellos trata de la detección de superficies planas de la escena en imágenes obtenidas a través de la cámara del dispositivo, ya que queremos aprovechar estas superficies planas para proyectar en ellas la información *aumentada* de la escena. El segundo módulo se ocupa de la proyección del modelo virtual en los sucesivos frames.

1.1. Contexto y motivación

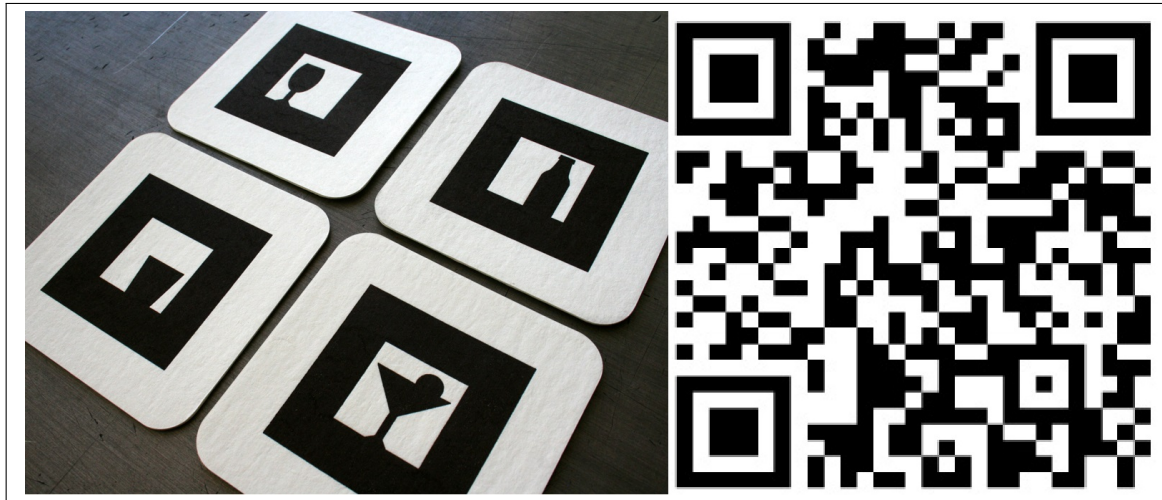


Figura 1.2: *Izda.* Marcadores con una imagen simple. *Dcha.* Código bidi.

Este proyecto se centra en realidad aumentada para dispositivos móviles basados en el sistema operativo *Android*. Se ha elegido Android en primer lugar por la mayor disponibilidad de documentación y el tamaño de la comunidad de desarrolladores, y en segundo lugar por la continuidad que luego se le pretende dar a los resultados obtenidos del presente estudio.

Este proyecto pretende investigar las posibilidades actuales de aplicación de realidad aumentada para dispositivos móviles, teniendo en cuenta tanto la potencia de estos dispositivos, como las herramientas ya existentes. Por *herramientas* entendemos tanto API's ya existentes propiamente de realidad aumentada como pueden ser Wikitude[1], Layar[2] o Vuforia[3]; como bibliotecas de visión por computador como OpenCV[4], SimpleCV[5] o FastCV[6].

Estas API's de realidad aumentada para Android. Se proporcionan en su mayoría bajo licencia, aunque se pueden encontrar licencias de prueba o incluso versiones gratuitas con determinadas limitaciones. Todas estas hacen uso de marcadores para la proyección de los modelos virtuales sobre la escena. Estos marcadores normalmente consisten en un código bidi o similar, o un cuadrado con alguna figura fácil de reconocer para algoritmos sencillos de búsqueda de patrones, como los que se muestran en la Figura 1.2.

Con la cámara del dispositivo se analiza cada uno de los frames en busca de cierto marcador predeterminado, y al reconocerlo, se proyecta el modelo virtual en su posición. El uso del marcador permite tener, en primer lugar, una referencia exacta de la posición en que debe proyectarse, pero también indica el ángulo con el que la figura virtual debe mostrarse, su orientación, y su tamaño. La Figura 1.3 muestra un diagrama de su

funcionamiento.

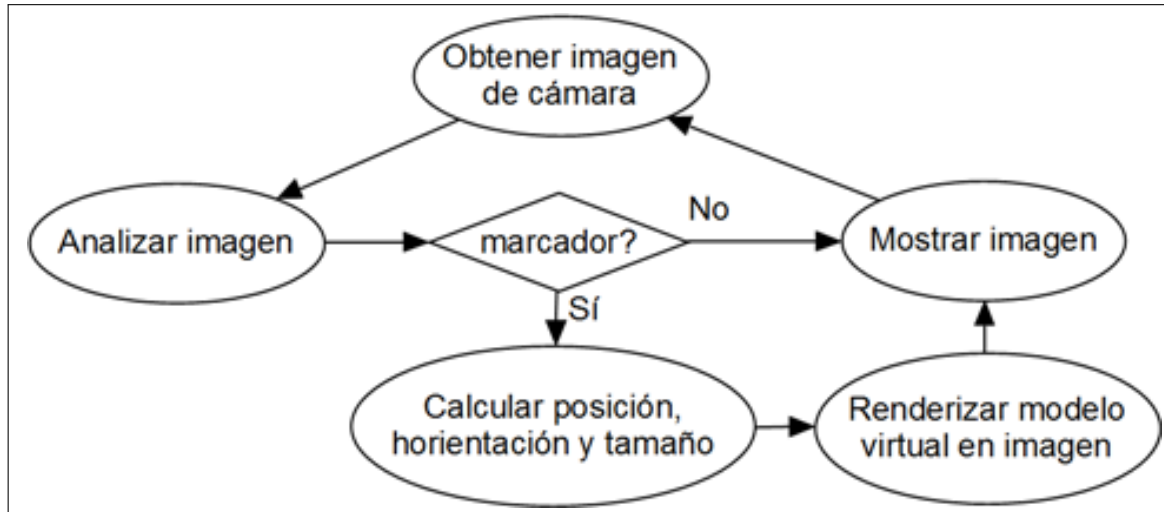


Figura 1.3: Diagrama general del proceso

El problema de este método es que la aplicación es completamente dependiente de este marcador. Es decir, solo va a funcionar si el usuario apunta con la cámara al lugar donde está el marcador. Esto limita sensiblemente su utilidad y lo relega casi a una simple mejora respecto a los códigos QR. Dado que, al fin y al cabo, la base de este método consiste en *encontrar* un patrón conocido, una mejora es suplir estos marcadores por imágenes reales. De esta forma en lugar de tener que *colocar* el marcador cuadrado en el lugar en donde queremos que aparezca nuestra imagen virtual, podríamos mostrarla, por ejemplo, sobre la fachada de un edificio emblemático, un cartel de publicidad o una tarjeta de visita. Esto nos permitiría, por ejemplo en un juego, que nuestro personaje pueda aparecer en cualquier ubicación real, no solo en donde coloquemos un marcador. La limitación en este caso es menor, ya que cualquier imagen que tenga suficiente textura para ser identificada podrá ser reconocida. Pero no hay nada que impida a la aplicación proyectar un modelo virtual sobre la copa de un árbol, sobre el horizonte, sobre un grupo de edificios lejanos, etc. Esto puede no parecer problemático, pero significa que en lugar de proyectar la realidad aumentada apoyada sobre una superficie, no hay ninguna restricción que impida proyectarla sobre cualquier punto de la escena, ya se trate de un punto de una pared, una rama de un árbol, o una nube en el cielo, pudiendo resultar en una impresión nada realista para el usuario.

1.2. Objetivos y alcance

El objetivo principal del proyecto es realizar un estudio de las posibilidades de la realidad aumentada y algoritmos de visión por computador en los dispositivos móviles actuales. Concretamente se han marcado los siguientes hitos:

1. Estudiar la viabilidad en un móvil de gama media de algunos algoritmos actuales de visión por computador.
2. Desarrollar un *framework* que permita reconocer una superficie plana con textura dentro de una escena captada con la cámara de un móvil.
3. Ser capaz de proyectar un modelo virtual sobre esta superficie plana de forma visualmente realista, ajustando su tamaño y su orientación a las del plano.
4. Crear un prototipo funcional que implemente los objetivos 2 y 3.

1.3. Metodología y entorno de trabajo

Este proyecto se desarrolla dentro del departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza. No obstante, este proyecto también pretende ser un apoyo y un punto de partida para la integración de lo aprendido en el futuro desarrollo de aplicaciones para la empresa *Qbitera Software*¹.

Esta empresa ha sido fundada por tres estudiantes de la Universidad de Zaragoza, y se dedica al desarrollo de software para dispositivos móviles. Pretence hacer uso de la tecnología para aplicar gamificación a la publicidad, y utilizar la realidad aumentada en distintos ámbitos como el turismo y la lectura.

Se ha desarrollado en Android utilizando las bibliotecas de *OpenCV*, *FastCV*, *Vuforia* y *OpenGL* principalmente.

El primer paso para el desarrollo de este proyecto es el estudio de las diferentes opciones que se pueden utilizar. Para la detección de superficies planas se ha implementado un prototipo que usa diferentes métodos y se han utilizado imágenes reales obtenidas con la propia cámara de los dispositivos de prueba utilizados durante el desarrollo. Se ha desplegado el prototipo sobre un *Samsung Galaxy Ace S5830* con SO *Android v2.3.6* y un *Samsung Galaxy Tab2 GT-P3100* con *Android v4.1.2*. También se ha utilizado el emulador de dispositivos virtuales de Google².

Una vez realizado este estudio y determinados los mejores métodos, se ha implementado un prototipo final que ha vuelto a ser sometido a las mismas pruebas de funcionalidad y rendimiento.

Después se ha estudiado la implementación comercial de Vuforia para proyección de realidad aumentada. Para ello se ha aplicado ingeniería inversa en las aplicaciones de muestra proporcionadas en el portal de desarrolladores[7]. Una vez comprendido su funcionamiento se ha pasado a estudiar la forma de modificarlo para adaptarlo al prototipo de detección de superficies anteriormente implementado. Vistos los problemas que se detallarán más adelante, se opta por implementar una solución intermedia utilizando el

¹<http://www.qbitera.com/>

²<http://developer.android.com/tools/devices/index.html>

algoritmo de flujo óptico de Lucas-Kanade, usando su implementación optimizada para dispositivos móviles de la API de FastCV, y OpenGL para la visualización gráfica.

Se incorporan, para mejorar el resultado, los sensores físicos presentes en el dispositivo móvil. Con ellos se obtiene la posición, orientación e inclinación de este, y por tanto, se calcula la orientación, inclinación y ángulo de proyección del modelo virtual para que se muestre *apoyado* sobre el plano objetivo.

Implementado el prototipo final, se realizan pruebas de nuevo con imágenes reales atendiendo a la correcta inicialización de la realidad aumentada. Es decir, que el modelo virtual aparezca dentro del plano objetivo, apoyado sobre este, y de un tamaño proporcional a él. En definitiva, se prueba que el resultado sea creíble.

Diagrama de Gantt

Aquí vemos el diagrama temporal propuesto al inicio del proyecto, y el real. Inicialmente se planificó el final del proyecto para ser presentado en junio. Pero diversos retrasos puntuales, y principalmente el problema encontrado en el módulo de proyección de realidad aumentada, propiciaron su retraso.

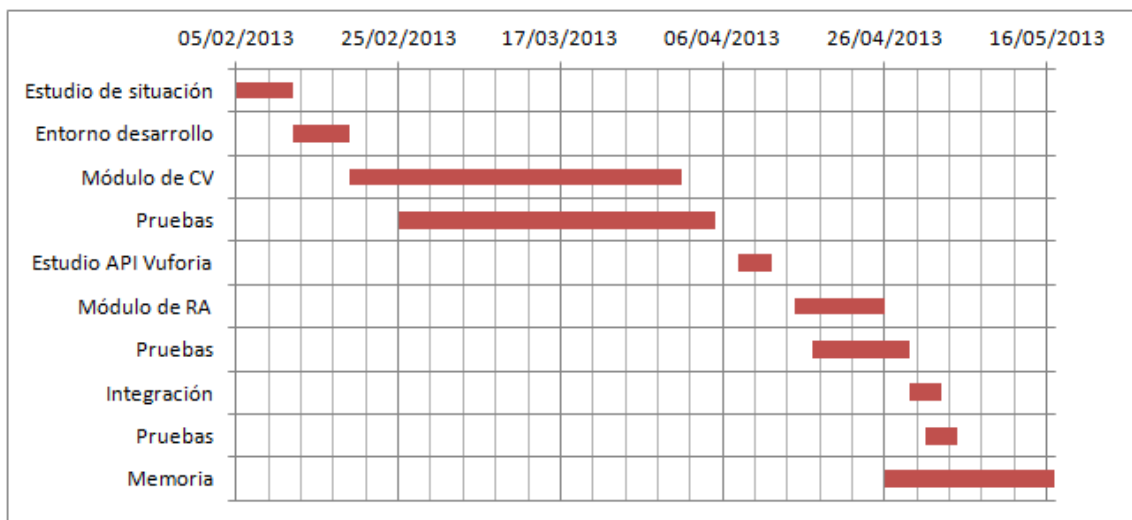


Figura 1.4: Diagrama de Gantt planteado al inicio del proyecto.

1.4. Trabajo previo relacionado

Este proyecto parte de la base de diversos estudios de métodos de visión por computador: el algoritmo *FAST Corner Detection* de Edward Roste[8][9][10], el algoritmo de flujo óptico de Bruce D. Lucas y Takeo Kanade[11] y su aplicación a la detección de objetos móviles en una escena[12], *oriented BRIEF* como alternativa a *SIFT* o *SURF*[13],

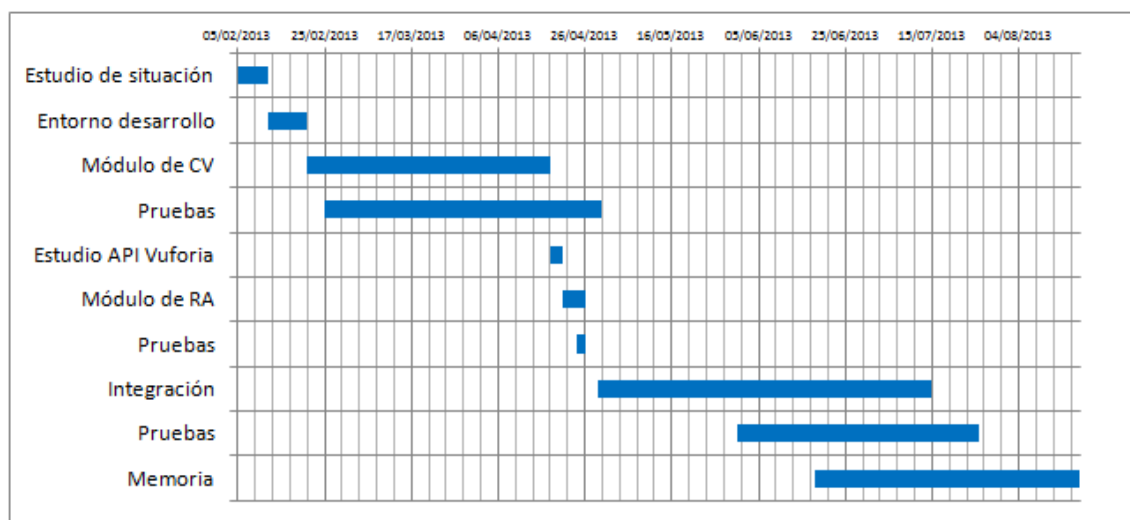


Figura 1.5: Diagrama de Gantt del desarrollo real del proyecto.

el método *BRISK*[14]. Para el desarrollo de los prototipos se ha partido del trabajo realizado por *Qualcomm* y sus plataformas de desarrolladores que ponen a disposición del usuario una serie de documentación y desarrollos de ejemplo. También se han consultado los proyectos de César Iñarrea Sagüés[15] que explica y desarrolla una aplicación similar a los ejemplos de Vuforia que utiliza cuadros como marcadores, y Ana Serrano Mamolar[16] que hace un estudio comparativo de distintos SDK de realidad aumentada. A este mismo efecto se ha consultado la publicación de Michael Mangan[17] de la conferencia *UPLINQ 2012* en donde compara *FastCV* con *OpenCV*.

1.5. Organización de la memoria

Este documento está dividido en cinco capítulos. El primero de ellos se corresponde con esta introducción en donde se explican las bases del proyecto. El segundo estudia el módulo de detección de superficies, donde se explican y analizan diferentes algoritmos de visión por computador utilizados. Dentro de este capítulo, en diferentes secciones, se detallan tanto la implementación de estos algoritmos estudiados como del módulo de detección de superficies en la aplicación final. También se explican todas las decisiones tomadas y las pruebas realizadas para tomar dichas decisiones. En el tercer capítulo se desarrolla el módulo de *tracking* y proyección del modelo virtual. Se analiza y explica la implementación utilizada, y el por qué de esta; así como los problemas encontrados y las soluciones adoptadas. En el cuarto capítulo se presentan los prototipos desarrollados. Y en el último capítulo se extraen y resumen las conclusiones obtenidas, se analiza el objetivo logrado y el alcance de este, contrastándolo con lo que se quería lograr. También se da una hoja de ruta para trabajos futuros sobre este proyectos y algunas pinceladas de posibles mejoras. Se incluye para finalizar la valoración personal del proyecto.

En los anexos se puede consultar la API desarrollada en C++ con los métodos de visión por computador utilizados en el proyecto en el anexo A. El funcionamiento del prototipo final de realidad aumentada se detalla a través de sus diagramas de estados en el anexo B. En el anexo C se puede consultar la batería de pruebas utilizada para testar el módulo de detección de superficies planas, y en el anexo D los resultados de las pruebas del módulo de proyección de realidad aumentada.

Módulo de detección de superficies planas en la escena

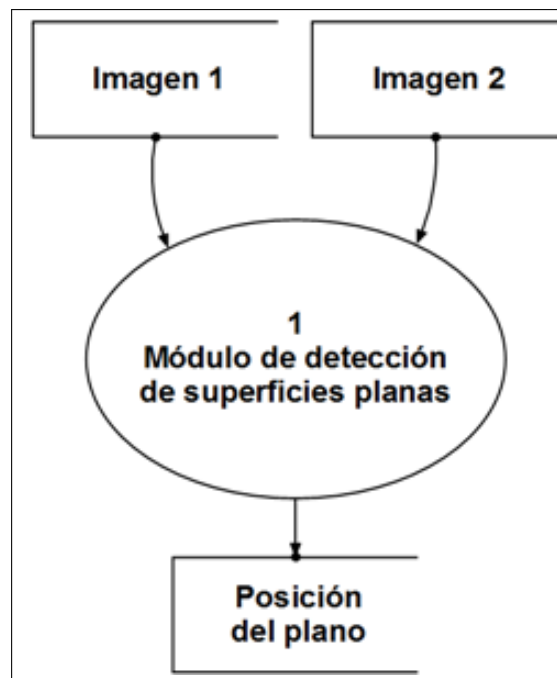


Figura 2.1: Módulo de detección de superficies planas

Como se ha visto en el diagrama general de la figura 1.1, el primer paso para llevar a cabo el objetivo del proyecto es la detección de una superficie plana en la escena sobre la que poder proyectar el modelo virtual. Para esto se han probado distintas técnicas de visión por computador que más adelante se describen, usando las APIs de OpenCV[18] y FastCV[19]. Es decir, este primer módulo recibe como entrada dos imágenes, o en el caso del prototipo desarrollado, *frames* capturados con la cámara, y los procesa ejecutando ciertos algoritmos de visión por computador. Su salida es una posición dentro de la escena

en la que se puede proyectar un objeto virtual, en el caso de que se haya encontrado una superficie plana adecuada para ello.

2.1. Análisis general

Para la detección de una superficie plana en la escena se ha optado por utilizar la aproximación que se describe en 2.2.

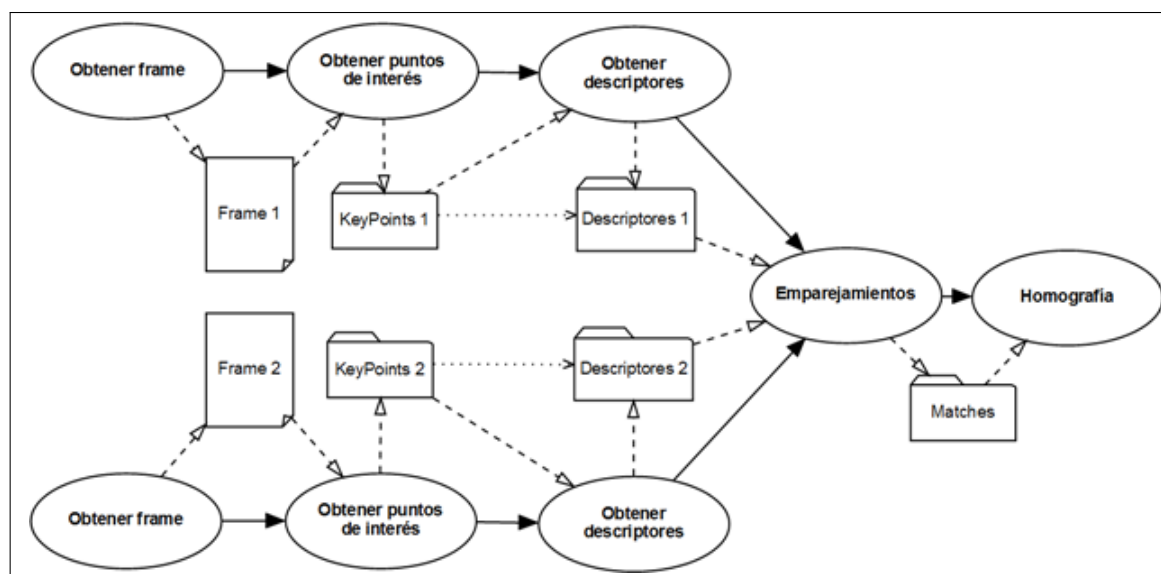


Figura 2.2: Diagrama general del proceso de detección usando dos frames

Se obtienen dos imágenes o, en el caso del prototipo desarrollado, dos frames cargados de la memoria interna del dispositivo o de una tarjeta *microSD* que estuviese insertada. Estos frames deben estar separados entre sí (una distancia mínima mayor o menor en función del método utilizado) y debe haber una rotación entre ambos. Se obtienen los puntos de interés o *keypoints* de ambas imágenes, y se extraen los descriptores de estos. Después se procede a buscar emparejamientos entre puntos de ambas imágenes, y con ellos se calcula una homografía.

Lo que se ha probado con este módulo han sido diferentes métodos para obtener puntos de interés, descriptores y emparejamientos para terminar calculando una homografía. Estos métodos se comparan teniendo en cuenta su tiempo de ejecución y el resultado obtenido con ellos.

La clave para detectar superficies planas en la escena es el cálculo de una homografía usando los puntos de interés de ambas imágenes, pues una homografía relaciona los puntos de dos imágenes que se encuentran en la misma superficie plana. Por tanto, si somos capaces de calcular una homografía entre los puntos de interés en dos imágenes separadas de la misma escena, presumiblemente habremos obtenidos un conjunto de puntos

coplanares. Calculando una homografía entre los emparejamientos de puntos obtenidos en las imágenes, podremos discernir cuáles de esos puntos corresponden a un mismo plano dominante en la escena: aquellos puntos que hayan votado por la homografía serán coplanares. Hablamos de plano dominante puesto que la homografía tenderá a obtener los puntos de la superficie mayor, en la que se habrán extraído mayor cantidad de puntos.



Figura 2.3: Puntos de interés obtenidos en dos *frames* de una secuencia de vídeo



Figura 2.4: Emparejamientos

En la imagen 2.3 vemos los puntos de interés que se han obtenido. Se marcan los puntos en los que hay una diferencia de gradiente, es decir, los cambios bruscos de color. Vemos además que se han obtenido puntos sobre la tarjeta que está en primer plano, sobre la que está medio oculta, pero también fuera, en las esquinas del monedero. Después de calcular la homografía se han representado en la figura 2.4 los puntos que han *validado* la homografía, esto es, los puntos que son coplanares. Se ve que se ha obtenido la superficie

dominante, es decir, la que estaba en primer plano, y los puntos fuera de esta han quedado descartados.

2.2. Detección usando una secuencia

En primer lugar se ha probado un método basado en múltiples frames de una misma secuencia. En tiempo real el usuario de la aplicación va a estar obteniendo frames de vídeo con la cámara de su dispositivo móvil, por lo que este parece un buen punto de comienzo. Para este método se ha utilizado el algoritmo de flujo óptico de *Bruce D. Lucas y Takeo Kanade*, en su implementación optimizada de *FastCV*.

El flujo óptico es el patrón del movimiento aparente de los objetos, superficies y bordes de una escena causado por el movimiento relativo entre el observador, en nuestro caso la cámara del dispositivo móvil, y la escena. Introducido inicialmente en la década de 1940 por el psicólogo estadounidense James J. Gibson, actualmente es utilizado en visión por computador para la percepción de movimiento, estimación de forma o distancia de un objeto respecto del observador.[20]

En todas las estrategias de estimación de flujo óptico se parte de la hipótesis de que los niveles de gris permanecen constantes ante movimiento espaciales en un tiempo dado. Dicha hipótesis da lugar a la ecuación general de flujo óptico (1), donde $I(x,y,t)$ corresponde a la intensidad en niveles de gris del píxel (x,y) de la imagen I en el tiempo t .

$$I(x,y,t) = I(x+dx,y+dy,t+dt) \quad (1)$$

Expandiendo (1) en series de Taylor sobre el punto (x,y,t)

$$I(x,y,t) = I(x,y,t) + dx \frac{\partial I}{\partial x} + dy \frac{\partial I}{\partial y} + dt \frac{\partial I}{\partial t} + \varepsilon$$

donde ε contiene la información de las derivadas de orden superior. Si se asume ε despreciable, la ecuación de flujo óptico puede reescribirse como

$$I_x u + I_y v + I_t = 0 \quad (2)$$

donde (u,v) , con $u = dx/dt$ y $v = dy/dt$, corresponde al vector de flujo óptico y, I_x y I_y son las derivadas parciales horizontal y vertical de la imagen, respectivamente. Para cada píxel x,y de la imagen, en el tiempo t , puede plantearse la ecuación (2), sin embargo, no existe una única solución para esta ecuación. Diferentes restricciones pueden emplearse para estimar el flujo óptico en la imagen. Lucas y Kanade se basan en gradientes espacio-temporales y asumen que el flujo óptico es constante sobre una región[12]. De esta forma, al ser constante en un vecindario cercano al píxel que se está analizando, se pueden

resolver las ecuaciones básicas del flujo óptico para todos los píxeles de ese vecindario por el criterio de los mínimos cuadrados.

Se ha utilizado la implementación del algoritmo piramidal de flujo óptico de Lucas-Kanade de *FastCV*, que se aplica de forma iterativa sobre sucesivos frames de una secuencia para realizar el seguimiento de los puntos de interés calculados en el frame inicial con un método de extracción de puntos de interés (en este caso puntos *fast*). Terminada la secuencia se calcula una homografía entre los puntos que han sido trackeados con éxito desde el primer frame al último, como se muestra en la figura 2.5. Es decir, se utilizan como emparejamientos para calcular una homografía los puntos del primer frame con los del último frame que han sido trackeados.

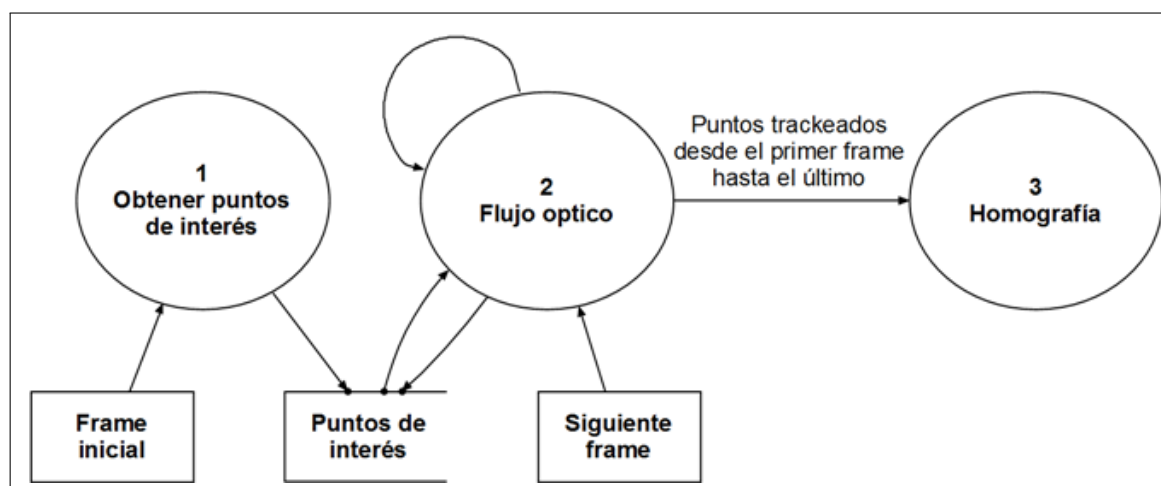


Figura 2.5: Diagrama del método

2.3. Detección usando dos frames distantes

En este caso el método utilizado se basa en dos imágenes de la misma escena separadas suficientemente. La idea es obtener dos imágenes en las que haya rotación del punto de vista, es decir, de la cámara del dispositivo, y obtener sus puntos de interés. A través de un método de *matching* se intenta hacer emparejamientos entre los puntos obtenidos en ambas imágenes para después obtener una homografía de la que poder deducir los puntos coplanares. En este caso, por tanto, es necesario que haya esta separación mínima entre las dos imágenes para evitar que dos puntos no coplanares puedan considerarse en el mismo plano por no haber cambiado la perspectiva.

Para realizar los emparejamientos se han probado dos algoritmos: la implementación de *FLANN* de *OpenCV* y la búsqueda de vecinos cercanos usando un *KDTree* de *FastCV*. *FLANN*, *Fast Library for Approximate Nearest Neighbors*, es una biblioteca que contiene una colección de algoritmos optimizados para la búsqueda de vecinos cercanos en grandes



Figura 2.6: En este caso no hay apenas separación entre los dos frames, y podría dar lugar a error al considerarse todos los puntos coplanares

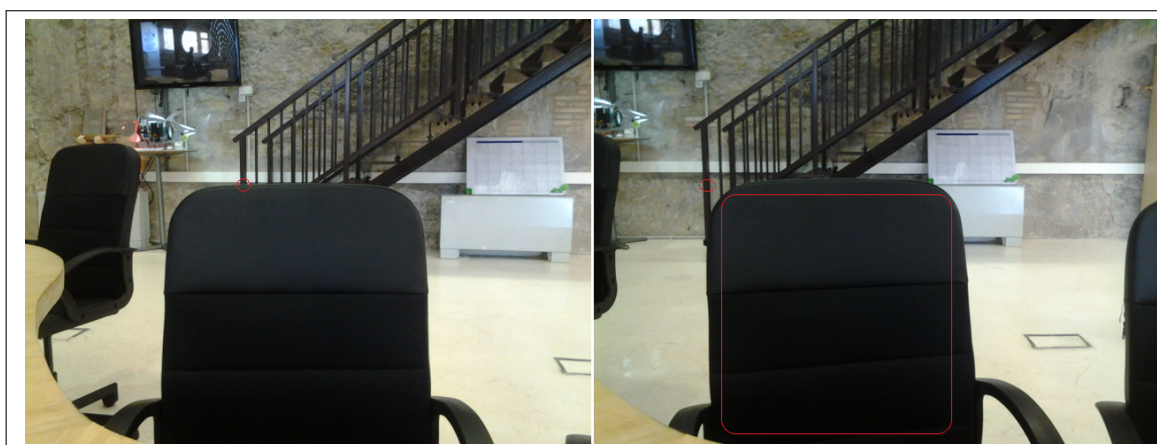


Figura 2.7: Entre los dos frames ha habido rotación, por tanto, no hay posibilidad de fallo en la homografía y los puntos de la barandilla no serán considerados coplanares con los del respaldo de la silla

conjuntos de datos. El método de *FastCV* consiste en crear un KDTree que se utiliza como base de datos en la que realizar consultas para encontrar los vecinos cercanos. En la base de datos se introducen tanto los valores correspondientes al descriptor de cada punto, como la inversa del cuadrado de la normal del vector que contiene esos valores. De esta forma, al hacer una consulta se busca primero utilizando este valor y después se refina el resultado comparando todos los valores del descriptor (un vector de 36 valores enteros de 8 bits cada uno) de los posibles aciertos.

La principal diferencia entre estos dos métodos es la misma que nos encontramos en

practicamente todas las implementaciones de *OpenCV* frente a *FastCV*. La primera es más robusta y está encapsulada de forma que se realiza un método complejo que hace diversas acciones, no todas utilizadas o requeridas en ese momento, con una sola llamada a función, mientras que *FastCV* proporciona el núcleo del método y corre por cuenta del desarrollador programar el algoritmo. De esta forma es más complejo y costoso, pero se puede implementar tan solo lo estrictamente necesario para realizar la función deseada. Los resultados obtenidos son una media aproximada de 100ms utilizando *OpenCV* frente a 45ms con *FastCV*. Se utiliza por tanto este último para la implementación del proyecto.

Para la extracción tanto de puntos de interés como de descriptores también se han comparado diferentes implementaciones de métodos de *OpenCV* y *FastCV* que a continuación se explican (en el capítulo de Pruebas se muestran los resultados). Para decidir qué métodos probar se ha partido de los resultados de estudios comparativos previos, como el realizado por Ievgen Khvedchenia[21], del que extraemos un resumen de los tiempos de ejecución de distintos algoritmos en la figura 2.8.

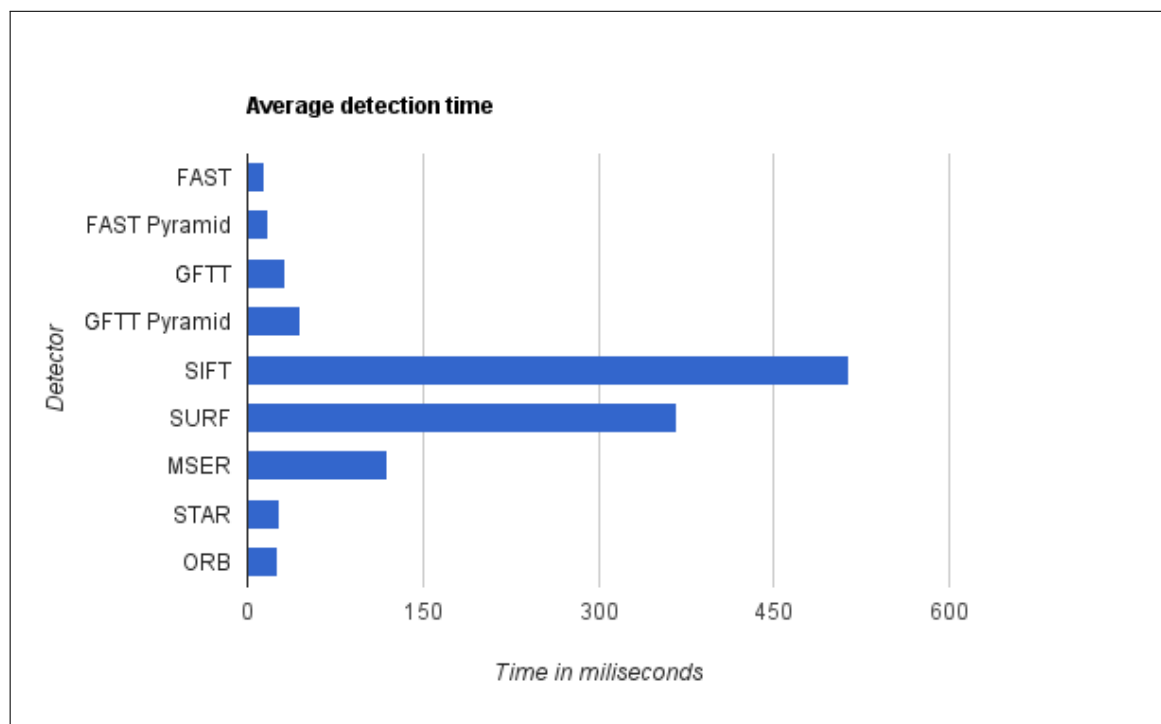


Figura 2.8: Tiempos de ejecución de distintos métodos de detección

OpenCV - FAST

Se trata de la implementación del algoritmo presentado en 2005 por *Edward Rosten* y *Tom Drummond*[8][9][10]. Este se basa en comparar un anillo de píxeles alrededor del punto en cuestión, como se ve en la imagen 2.9.

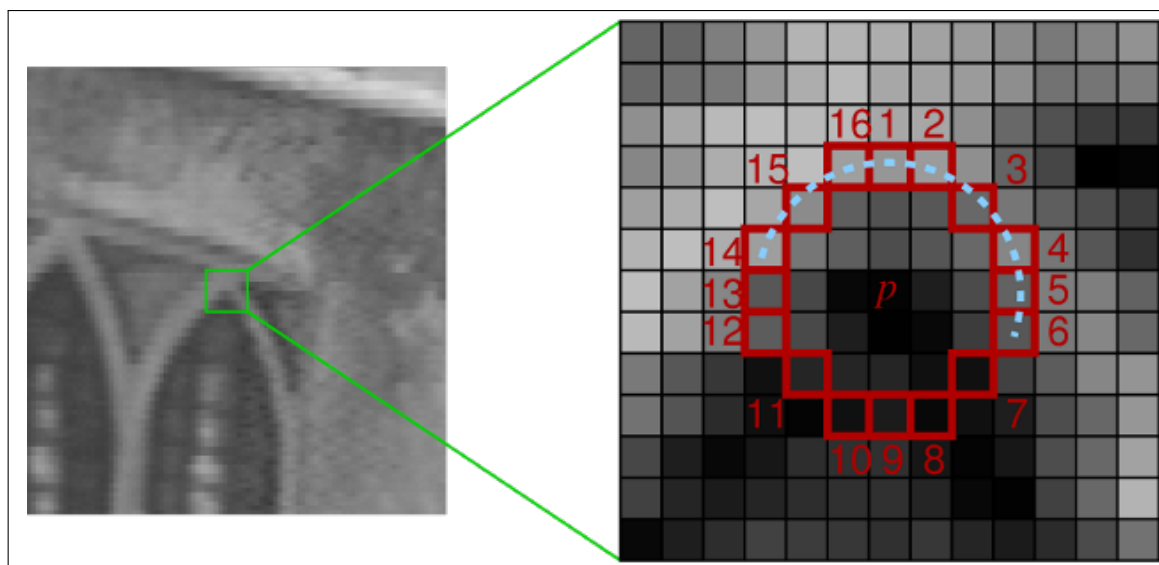


Figura 2.9: Path utilizado por FAST

FastCV - FAST

La implementación de *FastCV* permite personalizar algo más la ejecución del método. Mientras que *OpenCV* solo toma como parámetro el *threshold* a utilizar, es decir, el valor a partir del cual se considera un cambio válido de gradiente; esta permite especificar el número de píxeles que deben ser descartados en los bordes superior, inferior, izquierdo y derecho de la imagen, y el paso en píxeles entre las columnas de la imagen. Además proporciona un método de parada indicando el número máximo de puntos que se deben encontrar.

Este método de parada ha resultado efectivo en nuestro caso para evitar problemas de memoria en imágenes con, por ejemplo, mucha textura y mucha luz. En estos casos, alcanzado un determinado número de puntos de los que luego se extraían sus descriptores y se trataban de emparejar con un número similar de puntos de otra imagen, se alcanzaba el límite de memoria del dispositivo de pruebas, provocando una excepción manejada por el SO que causa la detención y reinicio de la aplicación prototipo.

OpenCV - Oriented BRIEF

Abreviado *ORB*, este método desarrollado en la incubadora californiana de empresas *Willow Garage* es resistente al ruido y al menos dos órdenes de magnitud más rápido que *SIFT*, mientras que su resultado es igual de bueno en la mayoría de las situaciones[13]. Se ha probado solo su implementación de *OpenCV* al no estar disponible en *FastCV*

Otros

También se han probado, con peores resultados, el método *BRISK* desarrollado por *Autonomous Systems Lab*[14], y las implementaciones de *OpenCV* de *SURF*, *SIFT*, *BRIEF* y *FREAK*[18].

Se han probado también en combinación con el adaptador *grid*. La imagen se divide en una cuadrícula y se aplica el método de detección de puntos de interés a cada celda de forma independiente.

2.4. Comparativa y pruebas

Para comparar los distintos métodos se han implementado en un prototipo, descrito en el capítulo 4, que carga las imágenes de la tarjeta SD o la memoria del dispositivo. Se aplica un método de extracción de puntos y descriptores, hace los emparejamientos y calcula una homografía. Finalmente muestra los resultados e indica el número de puntos de interés extraídos, el número de emparejamientos totales y los que han votado por la homografía, así como el tiempo de ejecución de cada parte del algoritmo.

Tras múltiples pruebas con imágenes aleatorias obtenidas con la cámara del dispositivo, se ha utilizado para las comparaciones una selección significativa de 11 escenarios con distintos grados de dificultad y particularidades representativas de los casos que se podrán encontrar en el uso real. A continuación se describen estos escenarios, cuyos nombres identifican cada banco de pruebas, que puede encontrarse en el anexo C y en el directorio correspondiente de la documentación digital adjunta.

1. Estatua_justicia. Dificultad *alta*. El monumento no tiene textura, al ser de un gris piedra casi uniforme, mientras que en el resto de la escena hay mucha textura tanto en los árboles como en los edificios con ventanas.
2. Cartel_aragon. Dificultad *alta*. El plano dominante se encuentra tras un cristal que, debido al cambio en el reflejo del sol según el distinto grado de incidencia en las distintas perspectivas, dificulta los emparejamientos entre puntos de interés.
3. Portal_reja. Dificultad *alta*. Ejemplo de patrones repetidos, lo que complica la obtención de emparejamientos válidos.
4. Cartel_corte_ingles. Dificultad *alta*. Ejemplo de escena en donde no hay un plano dominante, debido a la distancia del plano que se quería usar respecto de la cámara.
5. Cartel_ibercaja. Dificultad *alta*. En este escenario el supuesto plano dominante no tiene la textura suficiente debido al reflejo del sol. Además, hay diversos patrones repetidos que dificultan los emparejamientos.

6. Cartel_1E. Dificultad *media*. Este escenario representa el caso en que hay un claro plano dominante rodeado de elementos externos con muy alta textura, y otros planos secundarios.
7. Cartel_pared. Dificultad *media*. El plano que consideramos principal, el cartel, tiene suficiente textura, pero realmente se trata de un subplano alojado dentro del verdadero plano principal, la pared, que tiene otros elementos que destacan lo suficiente como para considerarse puntos de interés. Esto provoca que no sea fácil identificar el cartel.
8. Cartel_citi_bank. Dificultad *baja*. En este caso el plano dominante también se encuentra tras un cristal, pero al no estar expuesto directamente al sol y ser los reflejos producidos constantes en todos los frames. Además, no hay apenas textura fuera del cartel.
9. Cartel_niketos. Dificultad *baja*. Plano principal con suficiente textura, sobre una escena en la que apenas hay nada que pueda inducir a error.
10. Plaza_paraíso. Dificultad *baja*. En la escena hay dos posibles planos con textura, el cartel de *Plaza Paraíso* y el cartel de la tienda *Springfield*. Sin embargo, se obtienen más puntos de interés en el primero. Es donde el supuesto usuario estaría apuntando, y por tanto, está centrado en las imágenes.
11. Tarjeta. Dificultad *baja*. Escenario muy simple con dos superficies con similar grado de textura sin ningún elemento problemático. Ejemplo de cómo se selecciona la superficie del plano dominante de la escena.

Se han aplicado a todos ellos los 9 métodos implementados en el prototipo:

Implementaciones			
n	Puntos de interés	Descriptores	Emparejamientos
0	FAST	FastCV	KDTree
1	FAST	ORB	FLANN
2	FastCV	FastCV	KDTree
3	GridORB	ORB	FLANN
4	FAST	FREAK	FLANN
5	ORB compuesto	-	FLANN
6	ORB	ORB	FLANN
7	BRISK	BRISK	FLANN
8	GridBRISK	BRISK	FLANN

En el anexo C se pueden consultar los resultados detallados de todas las pruebas, así como imágenes mostrando los emparejamientos totales y los que han votado por la

homografía. La figura 2.10 presenta un gráfico resumido con los resultados globales. En este no aparecen los métodos basados en *BRISK* dado que son un orden de magnitud más lentos que el resto, y han sido directamente descartados por esta razón. Los mostrados en verde son los que han detectado la superficie plana con éxito en todas las pruebas en las que era posible hacerlo, el resto han tenido algún fallo.

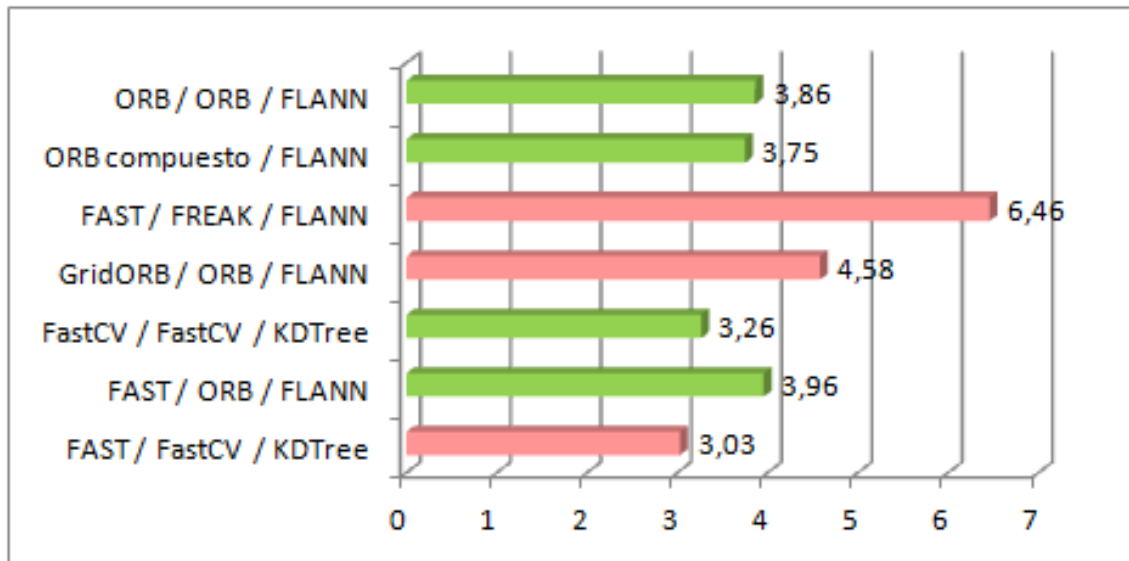


Figura 2.10: Comparación de los tiempos de ejecución medios de las distintas implementaciones, medido en segundos.

De las cuatro implementaciones que han obtenido los mejores resultados en cuanto a detección de la superficie en los distintos escenarios, la diferencia entre el mayor y menor tiempo de ejecución medio es de un 17.67%, 0.7 segundos. El mejor resultado se ha obtenido aplicando la implementación del algoritmo *FAST* de *FastCV* para calcular los puntos de interés y extraer los descriptores, y usar su método de emparejamiento basado en un *KDTree*.

Si atendemos a los resultados obtenidos, en la figura 2.11 vemos el número medio de puntos de interés obtenidos, emparejamientos, y emparejamientos válidos, es decir, los que votaron por la homografía que se calculó. El que se extraigan un mayor o menor número de puntos de interés no es una medida válida para evaluar la bondad del método a priori, pues un número excesivo de estos puede significar que se han extraído demasiados puntos redundantes que no aportan información de interés, pero si encarecen el coste de ejecución del cálculo de la homografía. Al obtener más puntos de interés, se obtienen más emparejamientos, y el tiempo de ejecución del cálculo de la homografía es proporcional al número de emparejamientos. El número de emparejamientos válidos, por tanto, sí que es una buena medida para comparar las distintas implementaciones.

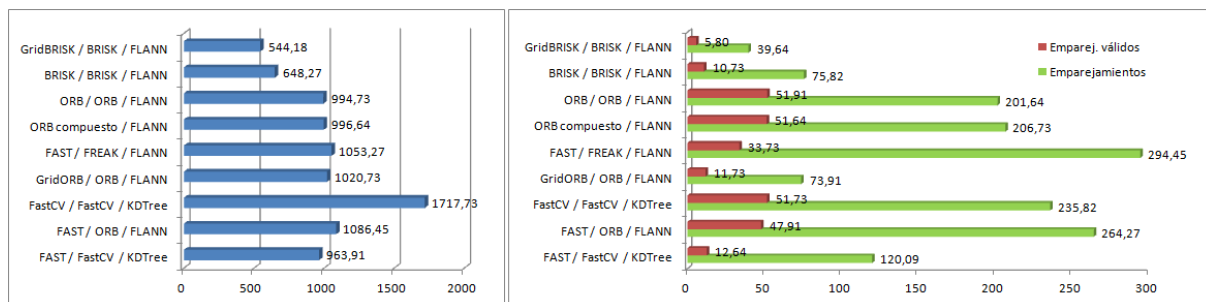


Figura 2.11: Izda. Puntos de interés obtenidos en media en todas las imágenes con las distintas implementaciones. Dcha. Emparejamientos realizados en media, y de estos, cuántos votaron por la homografía

Como puede verse, tanto las implementaciones de *ORB* como la de *FAST* de *FastCV*, han obtenido los mejores resultados en cuanto a número de emparejamientos válidos.

Por último, para terminar de valorar con cuál se obtienen los mejores resultados, se muestra en la figura 2.12 el porcentaje de emparejamientos válidos respecto a los totales en verde, y la relación entre estos emparejamientos válidos y el tiempo de ejecución total en que se han calculado, en azul.

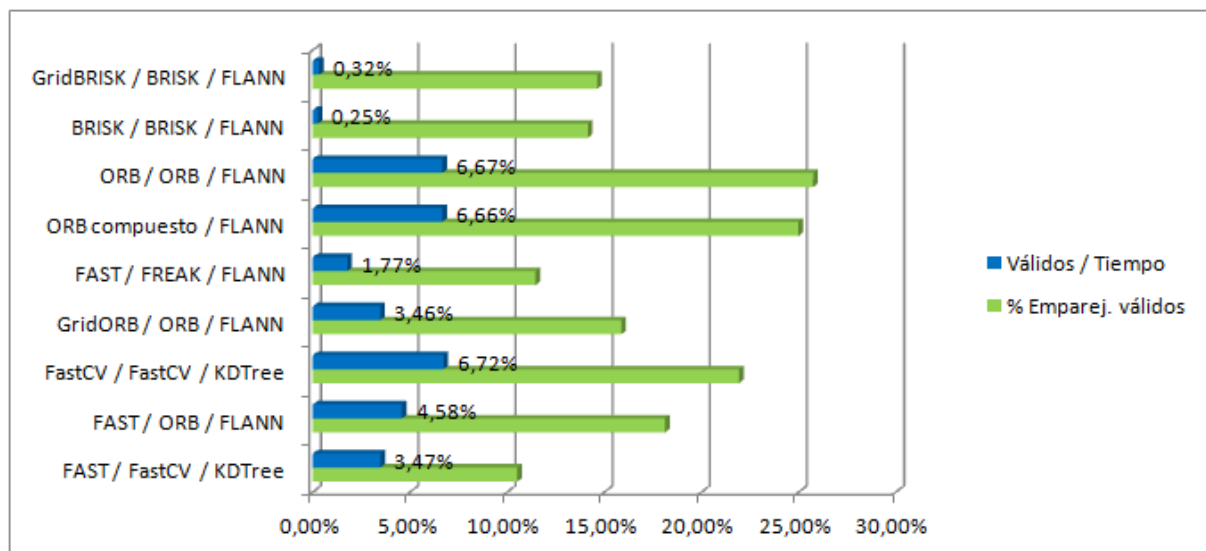


Figura 2.12: Relación entre emparejamientos totales, válidos y tiempo de ejecución

2.5. Conclusiones

Después de haber probado distintos métodos y compararlos atendiendo a su resultado y tiempo de ejecución, se ha tomado la decisión de utilizar la implementación de *FAST* de *FastCV* para obtener los puntos de interés de las imágenes, extraer los descriptores utilizando el método de *FastCV* y emparejarlos también a través de la implementación basada en *KDTree* de *FastCV*. La homografía se calcula utilizando la implementación de *OpenCV*. Como veíamos en la figura 2.12, los mejores resultados han sido los obtenidos de esta forma y con *ORB* y *FLANN*. Sin embargo, ante la aparente igualdad de resultados de ambos métodos, hemos optado por esta implementación por estar optimizada para arquitecturas basadas en el *Snapdragon* de *Qualcomm*. En una arquitectura *ARM* hemos obtenido estos resultados similares, pero en *Snapdragon* obtendríamos una mejora sustancial debido a esta optimización. Mientras que en el primero la API de *FastCV* hace uso de la CPU, en la segunda se utilizan los recursos de la CPU, GPU, VeNum (el motor de *NEON* de *Snapdragon*) y el DSP (procesador digital de señal, unidad con un conjunto de instrucciones y hardware específico para procesamiento de señales digitales, optimizado para aplicaciones que requieren operaciones numéricas a muy alta velocidad)[17].

En general, se obtienen muy buenos resultados sobre folletos o carteles sin demasiada textura alrededor, y unos resultados aceptables en planos encontrados en el mundo real, como por ejemplo carteles en la calle como se muestra en la imagen 2.13, siempre y cuando no haya en la escena árboles en una situación principal, u otros elementos con demasiada textura. En estos casos, si el número de puntos de interés obtenidos en estos otros elementos es elevado, puede hacer que el cálculo de la homografía se decante por estos puntos y deje de lado el plano dominante.

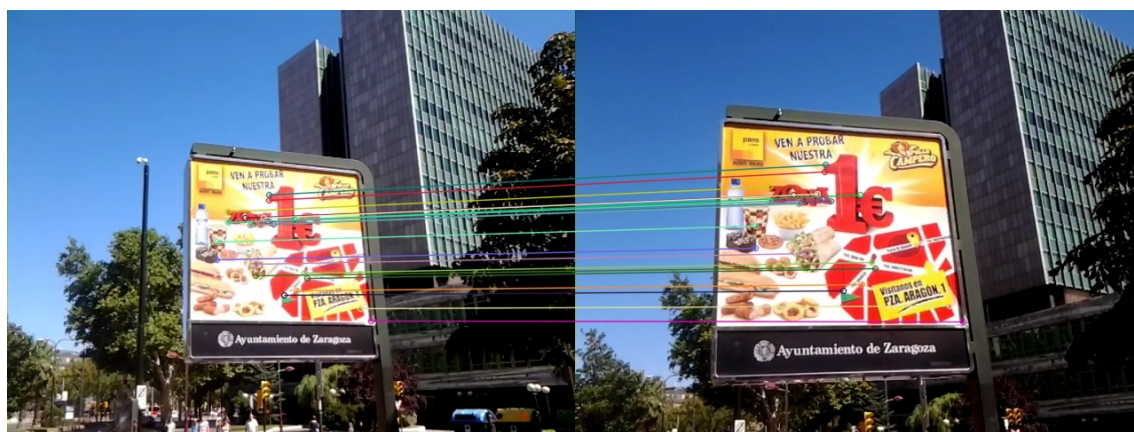


Figura 2.13: Emparejamientos válidos

Como se puede ver en las pruebas, al calcular una homografía es posible estimar una superficie plana basándose en los puntos de interés coplanares. Nos hemos encontrado sin embargo con algunos problemas.

- Las imágenes obtenidas de escenas en la que hay árboles suelen fallar. Esto es debido al gran número de puntos de interés extraídos en las hojas de los árboles, lo que provoca que, para la estimación de la homografía, tengan más peso las copas de los árboles que el supuesto plano dominante. Puede verse en algunos casos que incluso se *detecta* una superficie formada por los puntos de la copa del árbol, que al estar a una cierta distancia de la cámara, son considerados coplanares.

Una posible solución podría ser implementar un filtro que fuese capaz de detectarlos, y que no se considerasen válidos los puntos de interés obtenidos en estas zonas.

- En las imágenes hechas a un cartel colocado sobre una pared se presenta otro problema. Si el cartel tiene mucha más textura que la pared, es reconocido sin problema. No obstante, cuando la pared presenta mucha textura, como podrían ser ladrillos muy marcados, hendiduras, u otras formas similares, no es posible distinguir entre el cartel y el resto de la pared. Esto es debido a que, aunque el usuario considere el cartel como plano dominante, a efectos de la homografía toda la pared es un único plano, y los puntos de esta son coplanares con los puntos del cartel. Es decir, nos encontramos no ante un fallo del algoritmo, sino un fallo de interpretación del usuario.

En este caso, asumiendo que un cartel siempre va a tener una mayor concentración de puntos de interés (pues es una hipótesis bastante probable suponer que siempre tendrá algo más de textura que una pared), el modelo proyectado de realidad aumentada se mostraría en cualquier caso centrado sobre el cartel, pero su tamaño no va a estar limitado a este, sino que se calculará en función del tamaño del plano, es decir, la pared.

- El sol, y la iluminación en general, juega también un papel importante. La incidencia variable de este, u otras fuentes directas de luz, en los sucesivos frames de una escena pueden provocar que un mismo punto en dos frames distintos tenga unos valores de color distintos. Es decir, sus descriptores no son similares y no se emparejará correctamente.

Con todo esto concluimos que, salvando los problemas arriba mencionados, se ha cumplido el objetivo de detectar una superficie plana dominante en una escena capturada con la cámara de un dispositivo móvil. Hemos sido capaces de delimitar una zona apropiada dentro de la escena donde proyectar un modelo virtual de realidad aumentada. Aunque el tiempo requerido para ello en la media de todos los casos sea de algo más de 3s, el resultado es positivo puesto que se trata tan solo de la inicialización de la realidad aumentada. Es decir, el usuario solo deberá esperar este tiempo una vez a lo largo de la ejecución.

Módulo de tracking y proyección de realidad aumentada

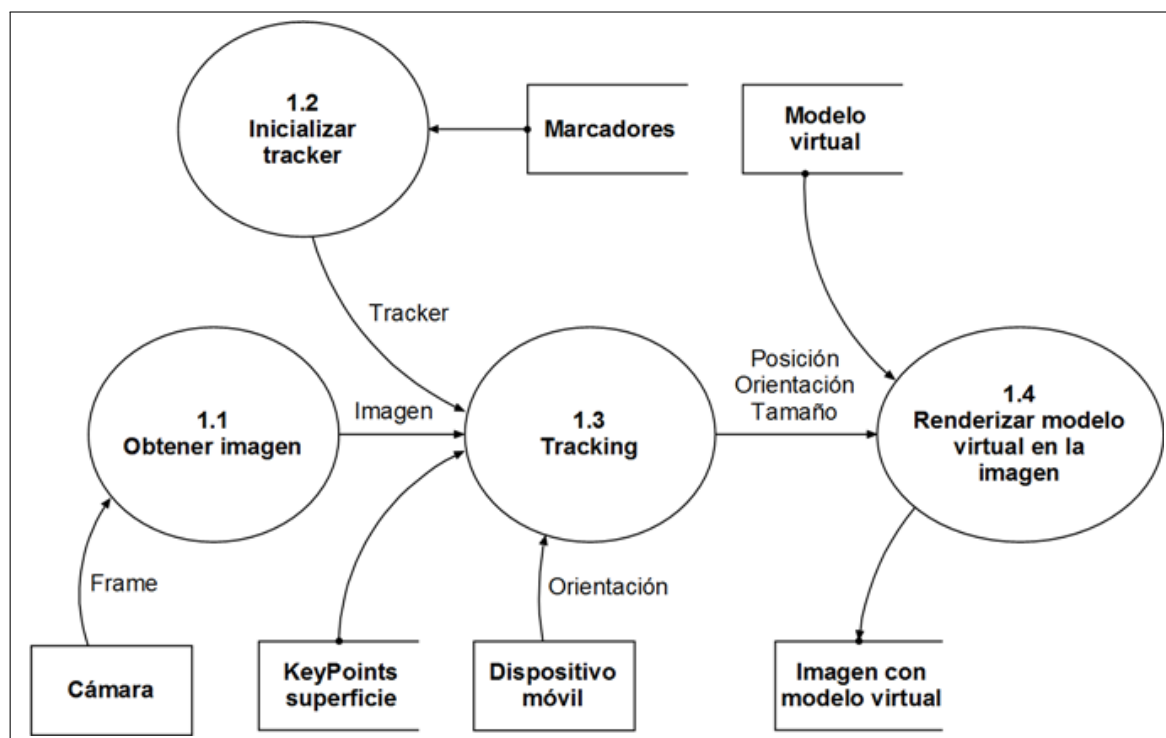


Figura 3.1: Diagrama de flujo de datos de nivel 1 del módulo de *tracking* y proyección del modelo virtual

Una vez encontrada y determinada la superficie plana en la escena, disponemos de información sobre ella con la que poder reconocerla y encontrarla en sucesivos frames. Es decir, podemos *trackearla*. Una vez hecho esto sabemos dónde está la superficie y podemos saber en qué posición relativa está dentro de la escena, por tanto, podemos proyectar sobre el plano nuestro modelo virtual de forma realista a la vista del usuario.

Esta es la funcionalidad del siguiente módulo del proyecto, que se describe en la figura 3.1.

3.1. Análisis general

Para este módulo se ha elegido el SDK de realidad aumentada de *Qualcomm*, *Vuforia*[22]. Se ha optado por esta plataforma y no otra en primer lugar porque el primer módulo ha sido desarrollado en parte usando *FastCV*, también propiedad de *Qualcomm*, por lo que la integración entre ambas plataformas es más sencilla. En segundo lugar, *Vuforia* cuenta con una comunidad de más de 60000 desarrolladores registrados[23]; y además se trata de una plataforma muy optimizada para su despliegue en móviles, con soporte tanto para *Android* como *iOS*, y también para el motor de videojuegos *Unity*[24].

Según su propia descripción, una aplicación de realidad aumentada basada en el SDK de *Vuforia* usa la pantalla del dispositivo móvil como unas "lentes mágicas". La aplicación renderiza la imagen previsualizada en la cámara y se superponen objetos virtuales 3D[25]. La figura 3.2 ilustra el proceso.

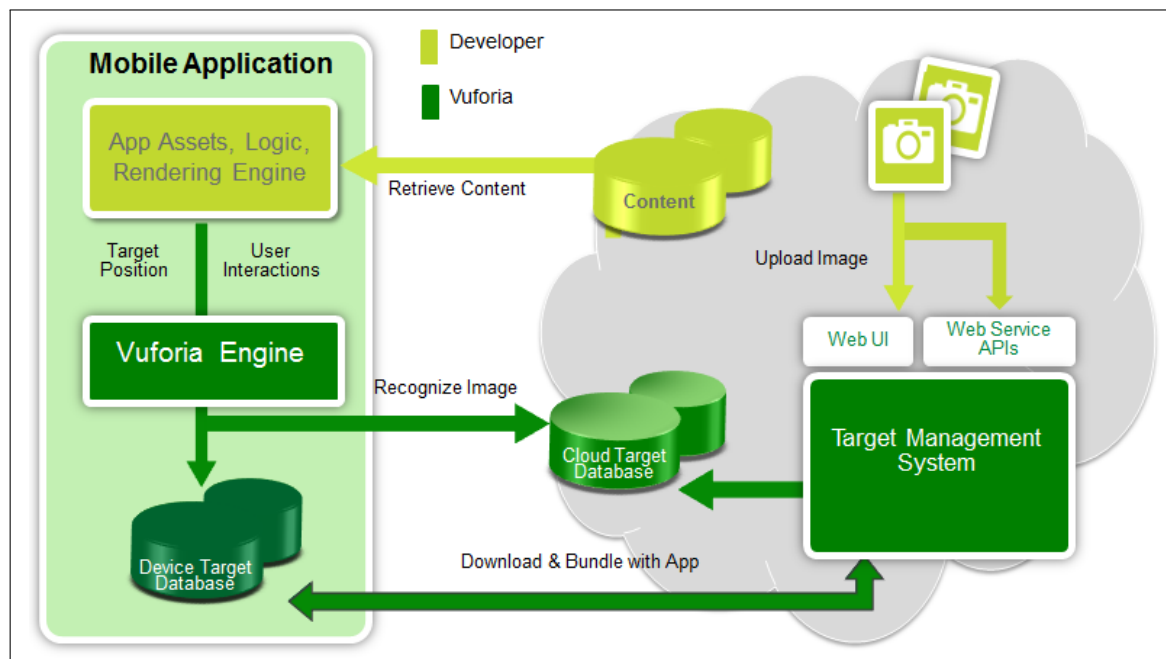


Figura 3.2: Uso de Vuforia

El desarrollador debe disponer previamente de las imágenes que quiere reconocer con su aplicación. Estas deben ser subidas al *Web Management System* de *Vuforia* y, o bien se crea una base de datos de imágenes alojada en la nube que se consultará en tiempo de ejecución, o bien se descarga dicha base de datos y se incluye en la aplicación. Durante la ejecución de la aplicación, esta interacciona con el motor de *Vuforia* para reconocer

las imágenes almacenadas en la base de datos. Cuando el motor de *Vuforia* detecta una imagen en la escena, se puede proyectar el contenido virtual creado por el desarrollador.

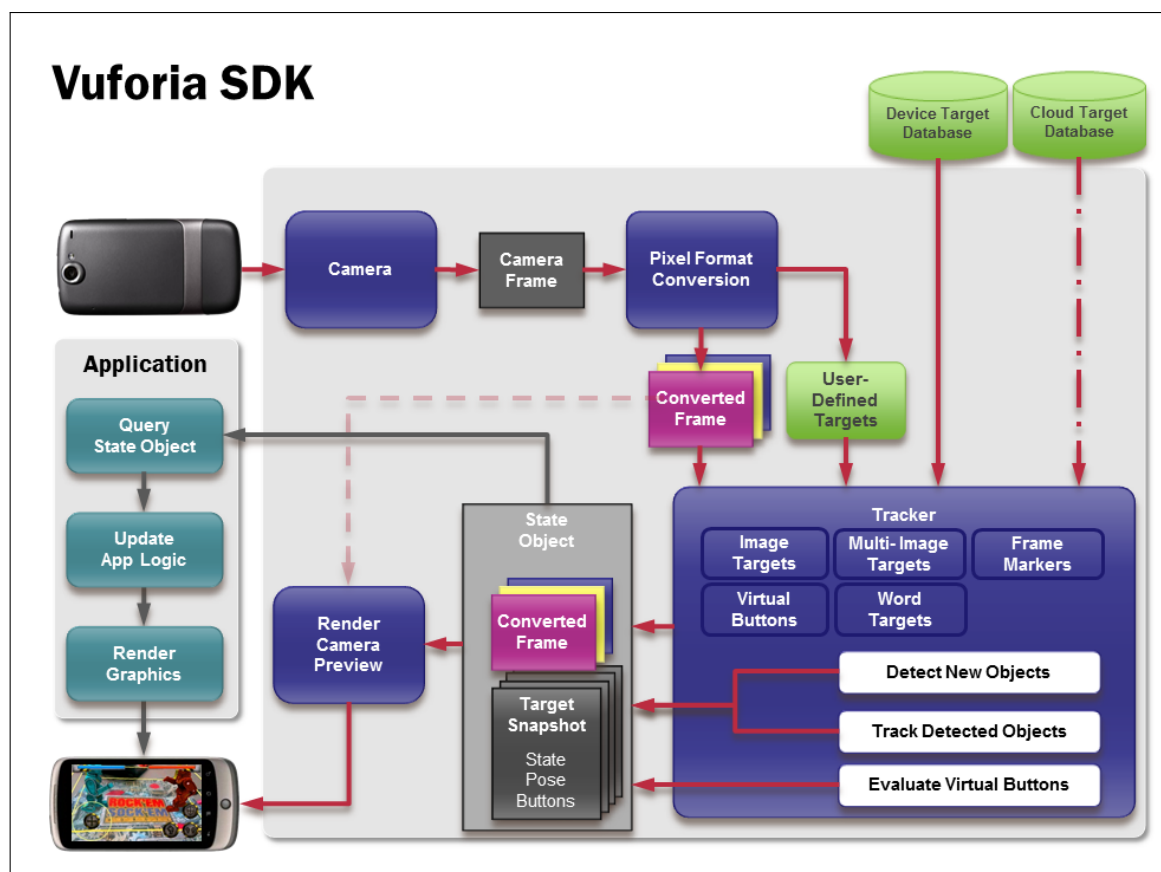


Figura 3.3: Diagrama de flujo de datos en una aplicación de ejemplo

La figura 3.3 muestra el diagrama de flujo de datos en una aplicación de ejemplo que usa el SDK de *Vuforia*. La cámara del dispositivo obtiene un frame que se convierte al formato requerido y se pasa al *tracker*, que puede tener distintas configuraciones. Este proyecto se ha basado en el *tracker ImageTargets*, es decir, reconoce un solo *target* o marcador en el frame. Esto es así porque la superficie plana se considera como un marcador-imagen, y suponemos que en la escena solo vamos a tener una única superficie plana dominante, por tanto no tendrá sentido poder reconocer dos marcadores, es decir, dos superficies planas distintas sobre las que proyectar nuestro modelo. El *tracker* obtiene la lista de marcadores que tiene que buscar de la base de datos alojada en la nube, o integrada en la aplicación, o se han podido crear los marcadores en tiempo de ejecución a partir de frames obtenidos por la cámara del dispositivo móvil. Esta última opción es la que se ha desarrollado en el proyecto. Finalmente, el *tracker* indica dónde debe renderizarse el modelo, y el nuevo frame se proyecta en la pantalla.

Basándonos en este diagrama, queremos modificar el proceso para obtener los marcadores no de una base de datos creada a través del *Web Management System* de *Vuforia*, sino obtenerlos del módulo de detección de superficies planas. Es decir, queremos utilizar como entrada del *tracker* la salida del módulo desarrollado anteriormente en este proyecto.

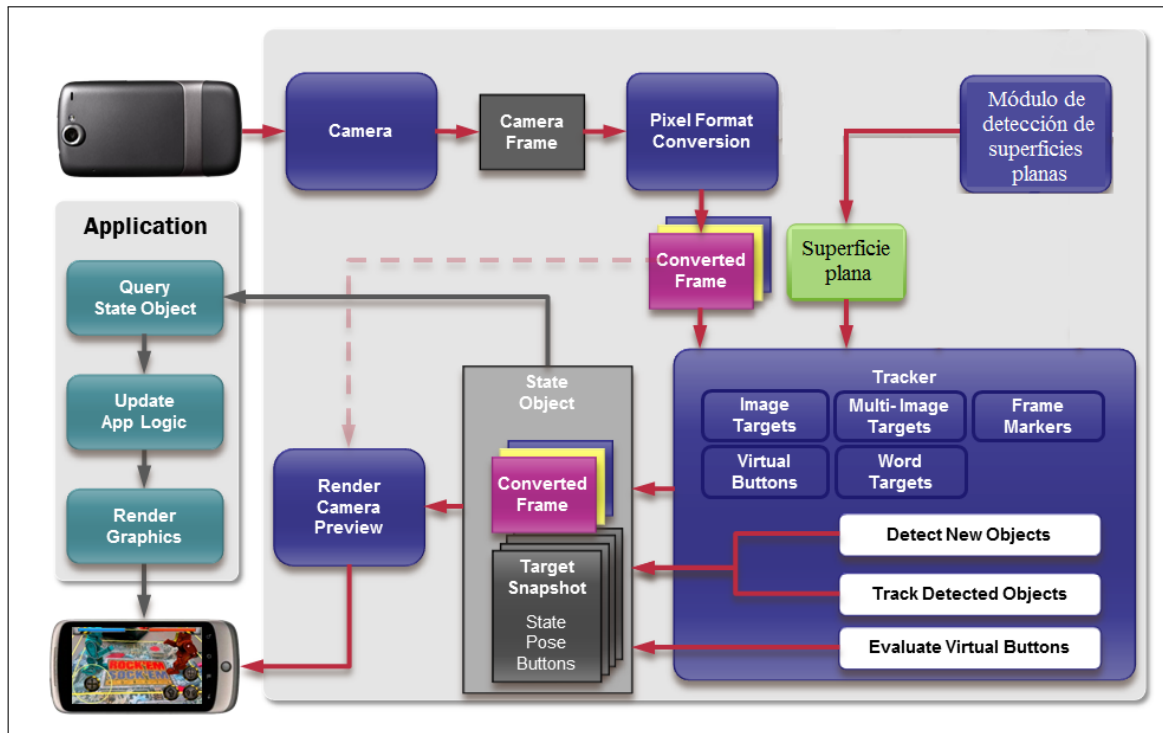


Figura 3.4: Modificación que se pretende llevar a cabo

3.2. Integración de una solución comercial: Vuforia

Como se ha explicado anteriormente, para la implementación del prototipo se ha utilizado el SDK de *Vuforia*. El punto de partida han sido dos de las aplicaciones de ejemplo proporcionadas a la comunidad de desarrolladores¹. La primera de ellas, *Image Targets*, reconoce como marcador una imagen de pequeñas piedras amontonadas que puede ser imprimida por el usuario, y proyecta una tetera tridimensional sobre esta. La segunda, *User-defined Targets*, permite al usuario seleccionar en tiempo de ejecución un frame de la cámara como marcador que será reconocido a partir de ese momento. A partir de este último se ha extraído el diagrama de estados general en que se pretende basar este módulo, mostrado en la figura 3.5. Se omite la parte irrelevante para el proyecto, como la preparación de estructuras y cargado de interfaz, texturas del modelo, etc. El diagrama

¹<https://developer.vuforia.com/resources/sample-apps>

comienza en el momento en que la cámara ha sido activada y se ha iniciado el análisis de las imágenes con una instancia del objeto *QCAR::Tracker*.

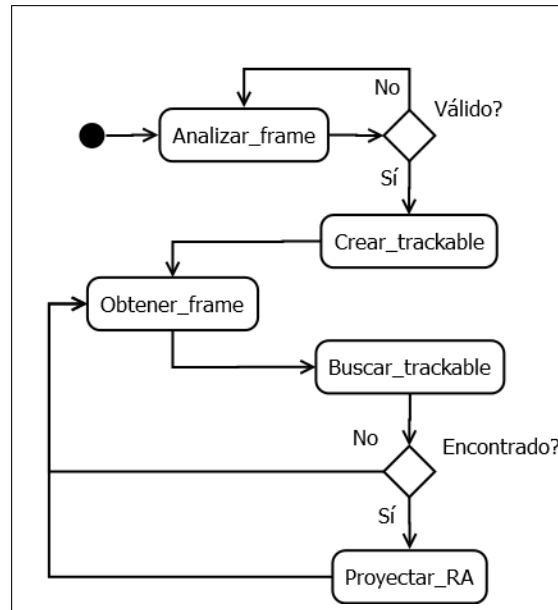


Figura 3.5: Diagrama de estados de la parte relevante de la aplicación de ejemplo *User-DefinedTargets*

Esta instancia del *tracker* es un detector de puntos de interés que analiza cada uno de los frames de la cámara en busca de *features*. Si en un frame se obtienen suficientes puntos, se considera válido para proyectar realidad aumentada sobre este y al usuario se le señala marcando un cuadro verde en la interfaz. En ese momento el usuario puede elegir pulsar el botón para aceptarlo como imagen de referencia. Se crea un objeto *Trackable* con la información del frame actual. Este será el objeto que será buscado a partir de ahora. Se entra en un bucle en el que se analizan todos los frames de la cámara y si se encuentra el objeto *Trackable*, se proyecta el modelo virtual.

Hay que destacar que este objeto *Trackable* no tiene que ser un objeto físico concreto o un plano real. Cualquier imagen en la que haya suficientes puntos de interés puede ser tomada como válida. A efectos prácticos, el frame con el que se construye el objeto se considera un plano, y el modelo virtual se proyecta *apoyado* sobre este, y de tamaño proporcional a la distancia de este.

Por tanto, la parte que debemos modificar es esta inicialización del *tracker*. Se quiere que en lugar de utilizar cualquier frame de cámara con suficientes puntos, se utilice la superficie plana que previamente hemos encontrado en la escena con el módulo anterior de este proyecto. La figura 3.7 muestra el diagrama de secuencia de creación de este objeto *Trackable*. El módulo que se quiere construir en este apartado debe tener un compor-

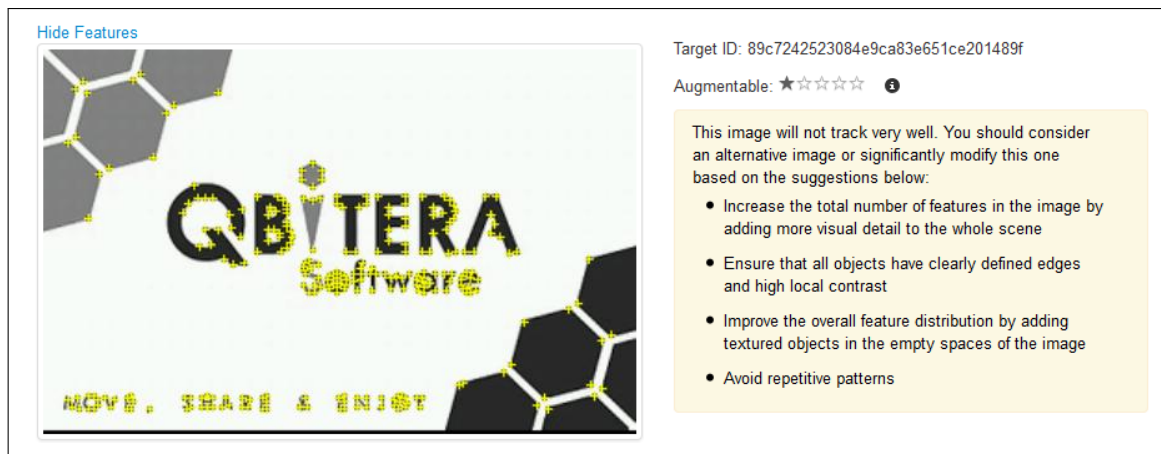


Figura 3.6: *Features* encontrados en una imagen cargada en el *Target Manager Web System*. Se muestra la puntuación de la imagen y las recomendaciones que debe seguir una buena imagen. Este análisis es similar al que se realiza en tiempo de ejecución en las aplicaciones con el objeto *QCAR::Tracker*

tamiento similar a la aplicación de ejemplo *User-defined Targets*, pero se debe eliminar por completo el análisis inicial de frames de *Vuforia* para sustituirlo por el módulo desarrollado en el anterior capítulo. Adicionalmente, se debe construir un objeto *Trackable* utilizando la salida de dicho módulo, en lugar de obtenerlo con el *ImageTracker*.

Problema encontrado

Esta solución no se ha podido aplicar al proyecto debido a la limitación existente de la API proporcionada. No hay que olvidar que *Vuforia* ofrece un servicio de pago de reconocimiento de imágenes en la nube, y a pesar de que ha liberado gran parte de la API con su SDK, de alguna forma tiene que proteger dicho servicio.

Actualmente es imposible crear un objeto *Trackable* de una forma distinta a la mostrada en la figura 3.7, a partir de un archivo de base de datos descargado e incluido en el código fuente de la aplicación creado con el *Target Manager Web System*, o utilizando su servicio de *Cloud Recognition* que permite mantener esta base de datos de *trackables* en su nube.

Citando a *AlessandroB*, desarrollador y moderador de *Vuforia*, a fecha de Junio 27, 2013, en su respuesta a la pregunta de si era posible de alguna forma crear un objeto *trackable* a partir de una imagen, o crear de alguna forma el archivo de base de datos:

Hi, such solution does not exist, you need to pass through the Target Manager.[26]

y ante la pregunta de si se contempla habilitar la posibilidad de crear *trackables* de otra forma en futuras versiones, la respuesta es:

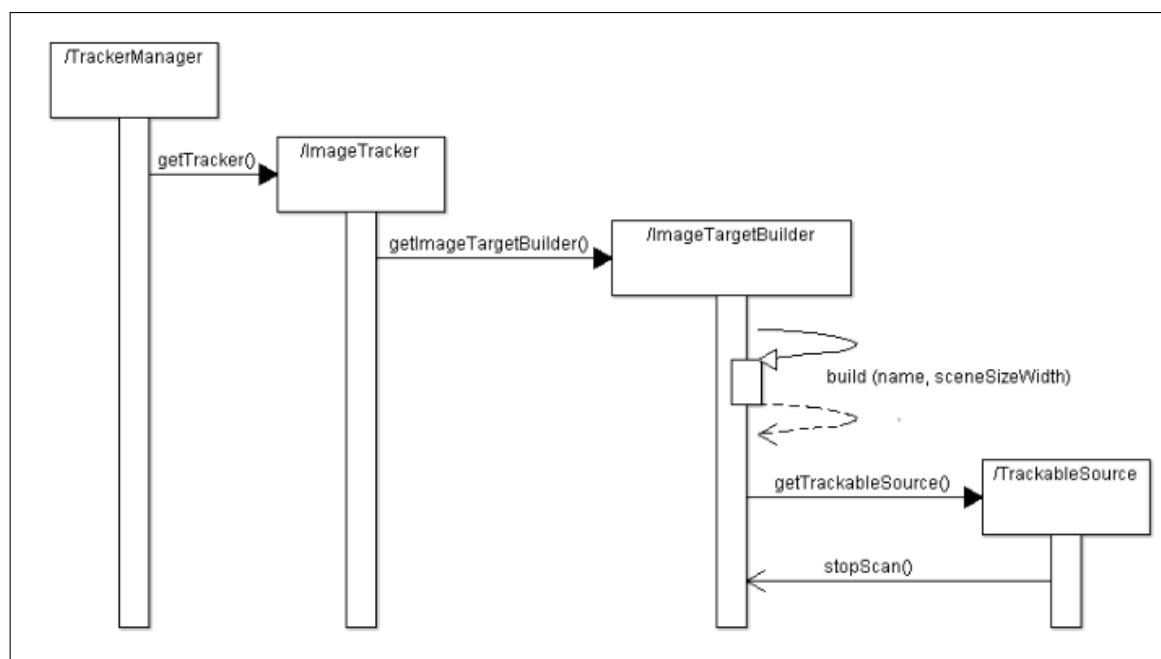


Figura 3.7: Diagrama de secuencia de la creación del objeto *Trackable*

Hi, at the moment I cannot tell whether this may change in the future. I can invite you however to post in our Forum Wish-List, as this might be taken into consideration for the future: <https://developer.vuforia.com/forum/general-discussion/wish-list> [26]

3.3. Solución provisional: Optical Flow

Al ser imposible implementar actualmente el módulo utilizando la solución comercial de *Vuforia*, pero con la posibilidad de que en un futuro sí que se libere esta funcionalidad, se ha buscado una solución provisional alternativa. La opción elegida ha sido aplicar el algoritmo de flujo óptico de *Bruce D. Lucas* y *Takeo Kanade* estudiado en el capítulo 2. Este algoritmo permite emparejar una serie de puntos de interés de dos imágenes siempre y cuando ambas estén separadas una distancia pequeña, del orden de unos pocos píxeles. Por tanto es ideal para realizar el seguimiento a un objeto en distintos frames consecutivos de una imagen de cámara.

Por tanto, para la implementación de este módulo se ha tomado como entrada el último frame obtenido para encontrar la superficie plana en la escena y sus puntos de interés que serán trackeados en los sucesivos frames. De esa forma se calcula la nueva ubicación del plano en cada nuevo frame, y usando los sensores del dispositivo móvil se obtiene la matriz de rotación, los ángulos y la orientación que permiten calcular de qué forma debe proyectarse el modelo virtual. Para calcular la posición en que debe centrarse

el modelo, se estima el punto medio del plano a partir de las coordenadas de la nube de puntos de interés del mismo.

En la figura 3.8 se muestra a alto nivel el funcionamiento del módulo. Se trata de un bucle que busca el plano, calcula la posición en que debe proyectarse el modelo virtual y renderiza. Los frames utilizados en el *Optical Flow* son siempre el actual y el anterior, donde ya se han calculado los puntos de interés y se ha localizado la superficie plana. Por tanto, en la primera iteración se le pasa como parámetro el último frame donde se detectó el plano en el módulo primero.

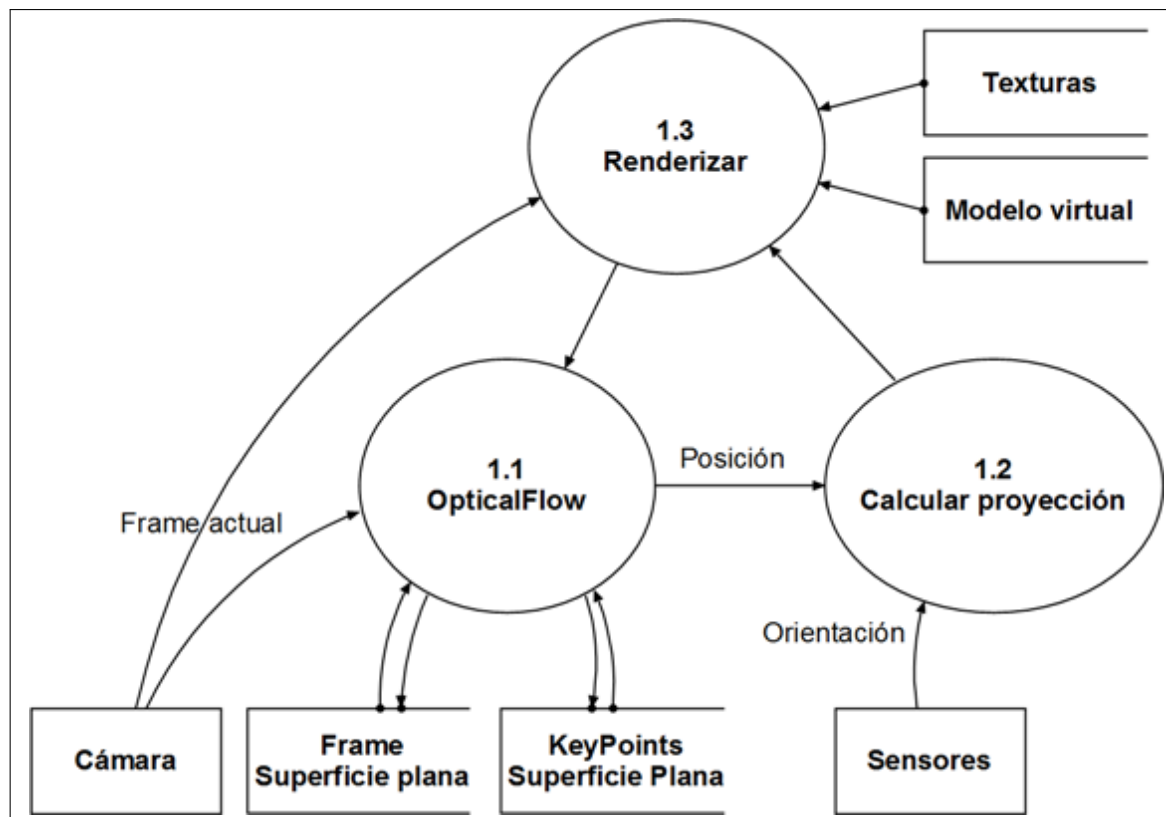


Figura 3.8: Diagrama de flujo de datos del módulo de *tracking* y proyección

3.4. Conclusiones

Como se ha explicado anteriormente, no ha sido posible implementar el módulo enteramente utilizando el SDK de *Vuforia* debido a una limitación que impedía crear un marcador en tiempo real a partir del resultado obtenido en el módulo primero. Por tanto, se ha tenido que optar por sustituir el *tracker* por otra solución desarrollada desde cero.

Podemos decir, sin embargo, que la principal conclusión del estudio de este módulo es que sí es técnicamente posible la proyección de realidad aumentada sin utilizar un marcador predefinido, sino una superficie plana encontrada directamente en la escena capturada con la cámara del dispositivo móvil. Aunque el prototipo desarrollado no es tan robusto como podría haber sido utilizando el SDK de *Vuforia*, sí que permite comprobar que es posible realizarlo. Y sabemos que dedicándole todo el tiempo y trabajo de desarrollo necesario, sería posible implementar desde cero una solución similar al *tracker* de *Vuforia*. Sin embargo, no es desarrollar un nuevo SDK de realidad aumentada el objetivo del presente proyecto.

Las imágenes 3.9 y 3.10 muestran dos ejemplos de visualización del modelo virtual, por defecto la habitual tetera tridimensional, sobre superficies planas. Estas son imágenes tomadas como pantallazos de la aplicación en tiempo real. Se pueden ver más pruebas en el anexo D

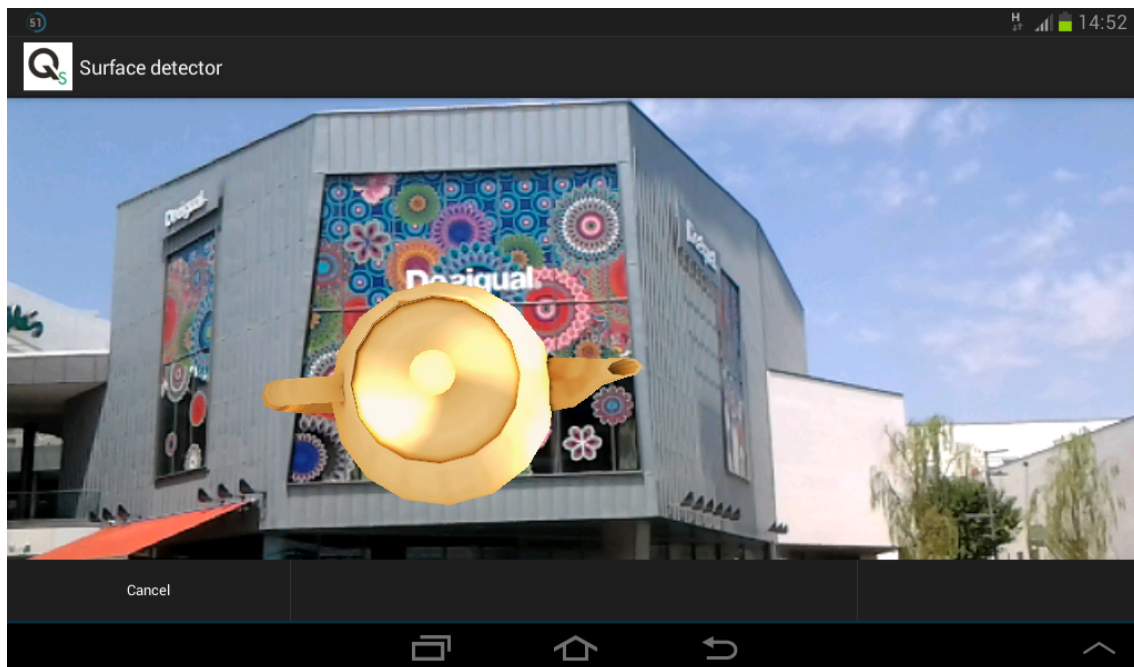


Figura 3.9: Visualización del modelo virtual renderizado sobre la superficie de un cartel publicitario.

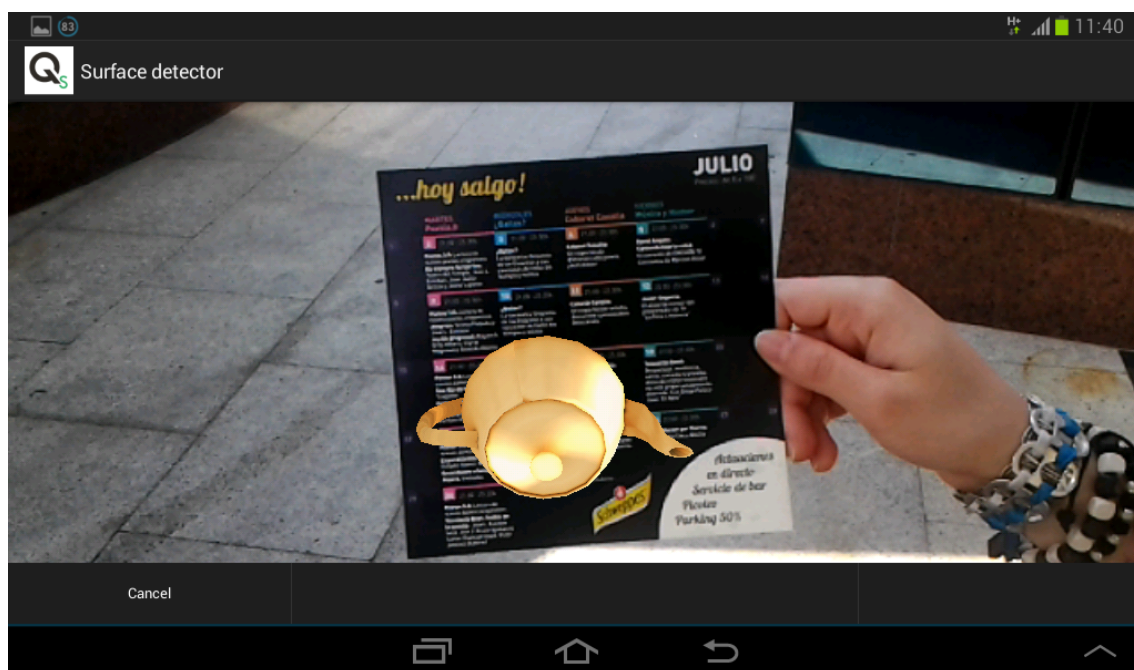


Figura 3.10: Visualización del modelo virtual renderizado sobre un folleto.

Como puede verse, la proyección es visualmente atractiva y realista. Queda *apoyada* sobre la superficie plana, y su tamaño se ajusta proporcionalmente al de dicha superficie. Además, la orientación es también proporcional a la orientación del plano y a la posición de la cámara. En la imagen 3.9, el plano es vertical respecto al suelo, y la cámara está situada por debajo del plano, es decir, se está viendo con una inclinación de unos 30° . El modelo virtual aparece proyectado acorde a estos valores, se muestra inclinado en contraposición a la inclinación de la cámara.

Lo mismo sucede en la imagen 3.10. En este caso la cámara se encuentra por encima del plano donde se proyecta el modelo virtual, así que la inclinación de este se ajusta para mantener la impresión de estar apoyado.

Capítulo 4

Prototipos

Se han implementado dos prototipos, uno como aplicación de prueba donde testar los distintos algoritmos de visión por computador y servir de framework para el módulo de detección de superficies planas, y un prototipo final donde se integra el primer módulo ya terminado con el segundo de *tracking* y proyección del modelo virtual. En la documentación adjunta se puede encontrar el código fuente de ambos prototipos así como sendos archivos *.apk* ya compilados y listos para ser instalados en un dispositivo *Android*.

Prototipo de prueba de algoritmos de visión por computador

Este prototipo consiste en una interfaz trivial que permite al usuario cargar dos imágenes de memoria interna o de una tarjeta SD introducida en el dispositivo, y aplicar los algoritmos en cuestión que se quieran probar sobre ellas. No se trata de una aplicación de usuario ni tiene ninguna funcionalidad concreta, es tan solo un framework en el que insertar el código del método que se desee testar. En la figura 4.1 se muestra el diagrama de flujo de datos del prototipo.

El prototipo está diseñado como un framework de forma que los métodos de obtención de puntos de interés, descriptores, emparejamientos y homografía puedan sobrescribirse e insertar el código que se desee testar. De esta forma, en el código fuente del prototipo, que se puede consultar en los anexos, se pueden ver en la interfaz nativa `/Prototipo1/jni/protectoCV.h` las siguientes funciones:

- *findPoints*(*n*, *method*): busca los puntos de interés de la imagen número *n* cargada previamente. Guarda estos puntos en la estructura correspondiente dentro de los parámetros del objeto, y devuelve el número de puntos encontrado.
- *findDescriptors*(*n*, *method*): busca los descriptores de los puntos de interés anteriormente calculados.
- *findMatches*(*method*): busca emparejamientos entre dos frames.
- *homography*(): calcula una homografía.

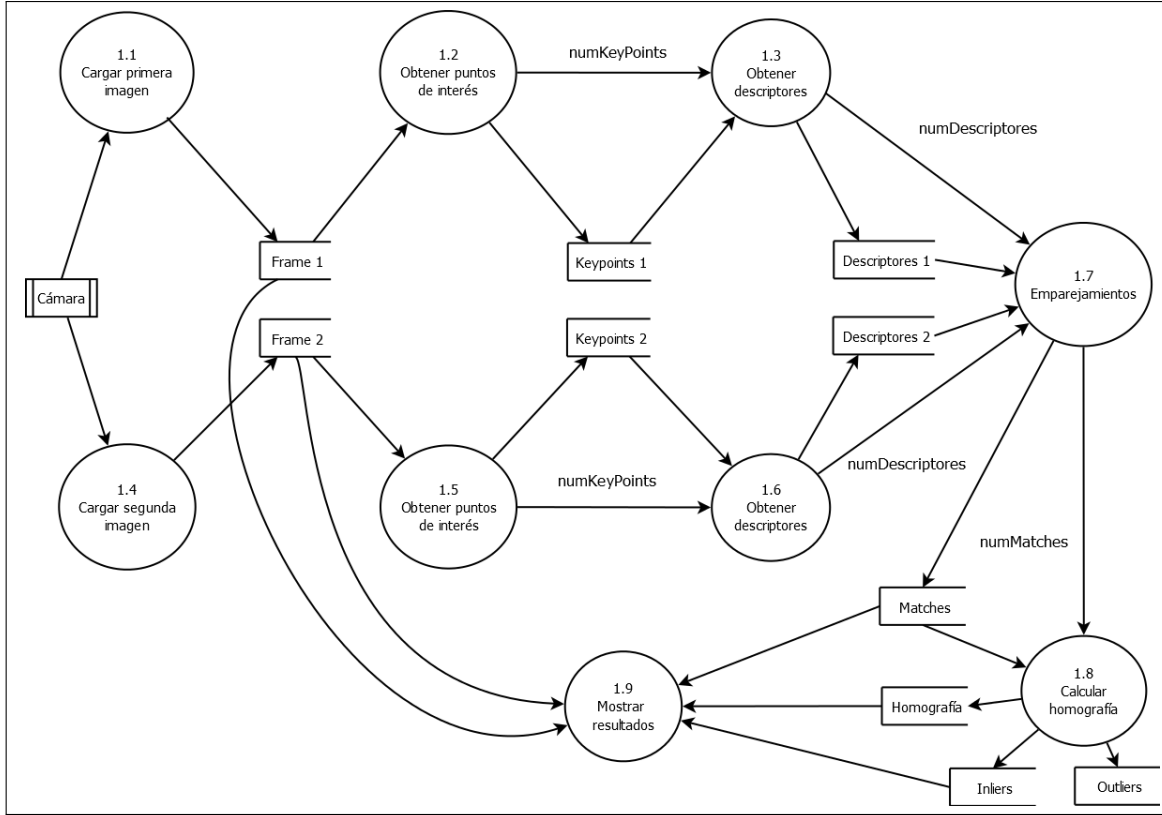


Figura 4.1: Diagrama de flujo de datos del prototipo de pruebas de algoritmos de visión por computador

Estas funciones deben cumplir la interfaz, pero puede modificarse su implementación para aplicar un método u otro de los descritos en el capítulo 2.

Prototipo final de RA

Este es el prototipo que integra ambos módulos y tiene la funcionalidad objetivo. Es decir, obtiene dos frames de la cámara del dispositivo móvil y los analiza para buscar una superficie plana. Después, esa superficie plana se *trackea* y se proyecta sobre esta el modelo virtual tridimensional.

La figura 4.2 muestra la interfaz de usuario. Se muestra en todo momento la imagen de la cámara, y en la barra de herramientas inferior se muestran solo tres acciones. En rojo se señala el botón *Cancel*, que terminará la ejecución de la aplicación. El botón señalado en verde capturaré el frame actual y lo usará como primero para iniciar el módulo de detección. Por último, en azul se muestra la opción de modificar el valor de threshold que se usará a la hora de obtener los puntos de interés, que desplegará el cuadro de texto que se ve en la imagen 4.3. Cuando el usuario pulsa en el botón de tomar foto, la interfaz

4. Prototipos

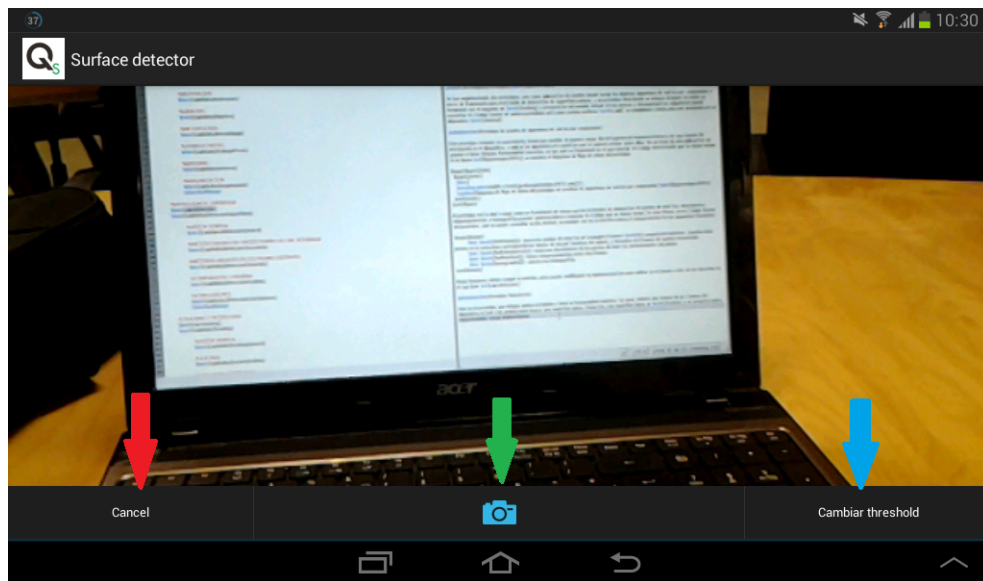


Figura 4.2: Captura de pantalla del prototipo. Se señalan los botones de la interfaz (de izquierda a derecha) de cancelar y volver al inicio, tomar el frame actual como primer frame que se usará para buscar una superficie plana, y modificar el valor del threshold aplicado.

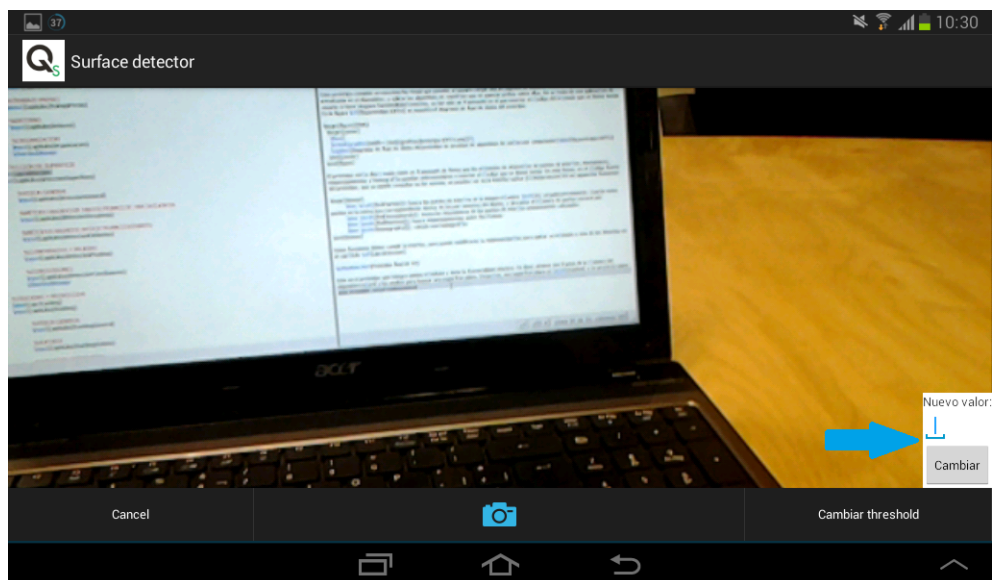


Figura 4.3: Captura de pantalla del prototipo. Se muestra el cuadro de texto que permite introducir un nuevo valor de threshold que será usado a partir de ese momento.

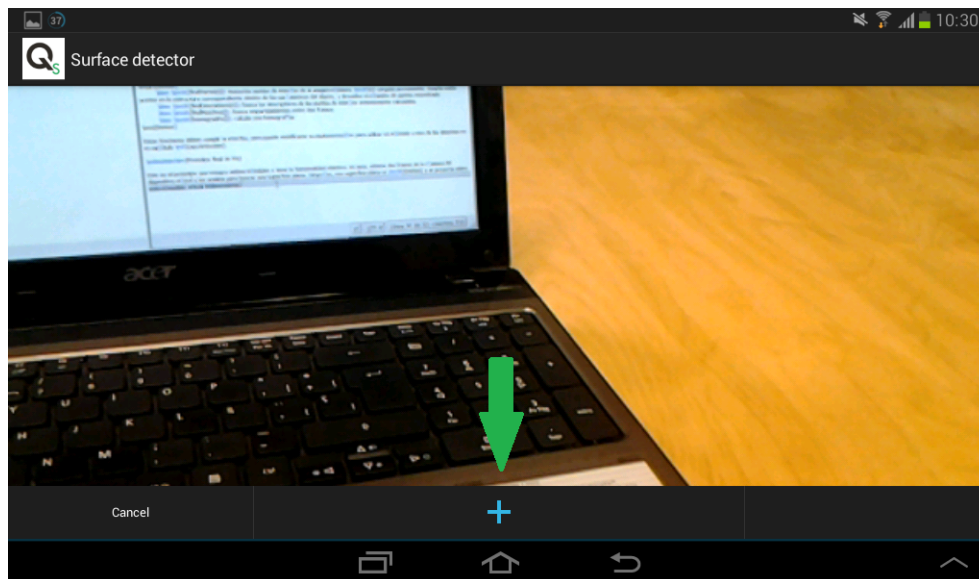


Figura 4.4: Captura de pantalla del prototipo. Se muestra el botón que toma el frame actual como segunda imagen que se analizará junto con la primera para buscar en ellas una superficie plana.

cambia a la que se puede ver en la figura 4.4, mostrando un único botón que captura el frame actual y lo utiliza como segundo frame para la búsqueda de una superficie. Si con ese segundo frame se consigue encontrar una superficie plana, se eliminará ese botón de la interfaz y se pasará al estado de realidad aumentada, trackeando la superficie y mostrando el modelo virtual sobre ella. Si no se ha podido encontrar con éxito la superficie en ese frame, se pueden seguir capturando frames hasta conseguirlo.

En el anexo B se describe el funcionamiento de este prototipo.

Conclusiones y valoración personal

Como se ha descrito a lo largo de este documento, se han conseguido los objetivos propuestos al inicio de este trabajo de forma satisfactoria.

Se han estudiado las bibliotecas de visión por computador de *OpenCV* y *FastCV* y se han implementado una serie de métodos utilizando ambas funcionando sobre *Android* con el objetivo de calcular una homografía que modele la proyección de puntos coplanares en una escena. Las funciones de *FastCV*, como era de esperar debido a la optimización de esta librería para móviles, han resultado ser más rápidas que las de *OpenCV*. Sin embargo esta última pone a disposición del desarrollador una mayor cantidad de métodos y estructuras de datos más complejas que facilitan la programación, frente a la casi total falta de documentación y estructuras de alto nivel de *FastCV*. El precio por esa optimización es que solo implementa las funciones más básicas de los algoritmos, por tanto en este proyecto se ha usado, y esto es práctica habitual en la comunidad de desarrolladores, una combinación de ambas. Mientras que para los métodos críticos se han usado las implementaciones de *FastCV*, para tratar los datos se han utilizado estructuras de *OpenCV*.

Se ha creado una API en C++ que implementa dichos métodos, preparada para usarse como parte de cualquier proyecto *Android* a través de la JNI, *Java Native Interface*. Una descripción de sus funciones puede consultarse en el anexo A.

Utilizando *Vuforia* y *OpenGL ES*, variante de la API diseñada para dispositivos integrados, se ha implementado un prototipo en *Java* y *C/C++* capaz de proyectar un modelo virtual sobre una superficie plana detectada en la escena en tiempo de ejecución. El modelo mostrado se ajusta al tamaño de la superficie y se orienta acorde a esta. No obstante, al encontrarnos con la limitación expuesta en el capítulo 3.2, que ha impedido usar el *tracker* de *Vuforia*, se ha tenido que implementar una solución provisional con el algoritmo de flujo óptico de *Lucas-Kanade*. El resultado por tanto no es tan robusto como sería deseable, como se podría conseguir integrando una solución comercial de visualización.

Se han estudiado soluciones comerciales para la visualización de elementos de realidad aumentada a las que técnicamente se podría integrar la detección de superficies planas desarrollada en tiempo de ejecución. Sin embargo, estas soluciones todavía son de carácter

cerrado, así que solo se han podido realizar pruebas por separado y no se puede aun integrar en un prototipo completo como el desarrollado en este proyecto.

5.1. Valoración personal

A nivel académico, la realización de este proyecto me ha supuesto la oportunidad de profundizar en el conocimiento de una plataforma completamente nueva para mí, el desarrollo para dispositivos móviles, y aprender a manejar sus sensores. Por otro lado me ha permitido tener una visión real y práctica de las aplicaciones de la visión por computador y la realidad aumentada, más allá de la teoría. Además, se han utilizado distintas herramientas de gestión de proyectos, como *Trello*; control de versiones y repositorios privados, *Bitbucket*; y utilidades para la creación de documentación y diseño como *Latex*, *Dia* o *ArgoUML*.

Desde un punto de vista personal, enfrentarse sin ningún conocimiento previo a una plataforma de desarrollo con sus particularidades, arquitectura, limitaciones y reglas nuevas y específicas, ha supuesto una oportunidad para poner en práctica lo aprendido en estos años.

5.2. Trabajo futuro

El principal camino a seguir como continuación de este proyecto es la sustitución o mejora del *tracker* implementado con el algoritmo de flujo óptico de Lucas y Kanade. Si, como han sugerido los desarrolladores, en una futura versión de *Vuforia* se liberase la capacidad de inicializar su *tracker* manualmente, tan solo habría que modificar el código del prototipo final implementado para utilizar dicho *tracker* en lugar del nuestro. De lo contrario, sería necesario implementar un *tracker* robusto del nivel de los ofrecidos como soluciones comerciales.

Otra línea de mejora secundaria es la optimización de la API implementada para la detección de superficies planas. Aunque la memoria de los dispositivos móviles es, al igual que en las máquinas convencionales, cada vez mayor y menos crítica a la hora de desarrollar aplicaciones; podría ser una buena mejora optimizar su consumo, pues este está directamente relacionado con el consumo de batería del dispositivo, algo que sí es un factor clave. Dado que se ha implementado con el propósito principal de ser utilizado en este proyecto, hay partes de código, estructuras y datos almacenados en memoria, que son solo útiles para el debugueado y para la demostración del funcionamiento de los prototipos, pero podrían ser eliminados en una aplicación real.

Bibliografía

- [1] Sitio web de Wikitude: <http://www.wikitude.com/>
- [2] Sitio web de Layar: <http://www.layar.com/>
- [3] Sitio web de Qualcomm: <http://www.qualcomm.com/solutions/augmented-reality>
- [4] Sitio web de OpenCV: <http://opencv.org/>
- [5] <http://www.simplecv.org/>: <http://www.simplecv.org/>
- [6] Sitio web de FastCV: <https://developer.qualcomm.com/mobile-development/mobile-technologies/computer-vision-fastcv>
- [7] Aplicaciones de muestra de Vuforia: <https://developer.vuforia.com/resources/sample-apps>
- [8] *Fusing Points and Lines for High Performance Tracking*, Edward Rosten y Tom Drummond. Department of Engineering, University of Cambridge, Cambridge, CB1 2BZ, UK, 2005
- [9] *Machine Learning for High-Speed Corner Detection*, Edward Rosten y Tom Drummond. Department of Engineering, University of Cambridge, Cambridge, UK, 2006
- [10] *Faster and Better: a machine learning approach to corner detection*, Edward Rosten et al. Department of Engineering, University of Cambridge, Cambridge, UK, 2008
- [11] *An Iterative Image Registration Technique with an Application to Stereo Vision*, Bruce D. Lucas y Takeo Kanade. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1981
- [12] *Detección de objetos móviles en una escena utilizando flujo óptico*, David Mora et al. XIV Simposio de tratamiento de señales, imágenes y visión artificial, STSIVA 2009
- [13] *ORB: an efficient alternative to SIFT or SURF*, Ethan Rublee et al. Willow Garage, Menlo Park, California, CA 94025, USA

-
- [14] *BRISK: Binary Robust Invariant Scalable Keypoints*, Stefan Leutenegger et al. Autonomous System Lab, ETH Zürich, 2011
 - [15] *Desarrollo de una aplicación de realidad aumentada mediante la arquitectura Vuforia para la obtención de información de cuadros*, César Iñarrea Sagüés. Escuela Técnica Superior de Ingenieros Industriales y de Telecomunicación, Navarra, España, 2012
 - [16] *Herramientas de desarrollo libres para aplicaciones de realidad aumentada con Android. Análisis comparativo entre ellas*, Ana Serrano Mamolar. Universidad Politécnica de Valencia, España, 2012
 - [17] *Harness the Rise of the machines: FastCV, Accelerating Mobile Computer Vision*, Michael Mangan, UPLINQ 2012 Conference
 - [18] API de OpenCV: http://docs.opencv.org/modules/features2d/doc/common_interfaces_of_feature_detectors.html, Common Interfaces of Feature Detectors
 - [19] API de FastCV: <https://developer.qualcomm.com/docs/fastcv/api/modules.html>
 - [20] Wikipedia: https://en.wikipedia.org/wiki/Optical_flow, Optical Flow
 - [21] *Comparison of the OpenCV's feature detection algorithms*, Ievgen Khvedchenia, 2011. Disponible: <http://computer-vision-talks.com>
 - [22] API de Vuforia: <https://developer.vuforia.com/resources/api>
 - [23] Sitio web de Vuforia: <https://www.vuforia.com/>, Vuforia's Growing Ecosystem
 - [24] Sitio web de Unity: <http://unity3d.com/>
 - [25] Guía de desarrollo de Vuforia: <https://developer.vuforia.com/resources/dev-guide/getting-started>, Developing with Vuforia
 - [26] Foro de desarrolladores de Vuforia: <https://developer.vuforia.com/forum/>
 - [27] *Lucas-Kanade 20 Years On: A Unifying Framework*, Simon Baker et al. The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA
 - [28] *Computer Vision Working Group Proposal, 1st December 2011*, disponible: <http://www.khronos.org/assets/uploads/developers/library/Computer-Vision-Working-Group-Proposal-Dec11.pdf>

Apéndice

Apéndice A

API del framework de visión por computador desarrollado en C++

*int loadImage (JNIEnv *env, jobject obj, jstring root, jint n);*

@brief

Lee una imagen y la guarda para su posterior tratamiento

@param

root: ruta de la imagen

n: número de imagen

@return

0 si hay algún error

*void saveMatches (JNIEnv *env, jobject obj, jstring root);*

@brief

Guarda la imagen de los matches encontrados. Se supone que previamente se habrá llamado a la función showMatches()

@param

root: ruta donde se guardará la imagen

void showImage (JNIEnv *env, jobject obj, jint n);

@brief

Renderiza la imagen

@param

n: número de imagen

void showCorners (JNIEnv *env, jobject obj, jint n);

@brief

Renderiza la imagen indicada por n con los puntos de interés que previamente se han calculado

@param

n: número de imagen

void showMatches (JNIEnv *env, jobject obj, jint method);

@brief

Muestra una composición de la imagen inicial y final de la secuencia, mostrando los matches que ya se han calculado

@param

method: 0 para *OpenCV FLANN* o 1 para *FastCV KDTree*

jint findPoints (JNIEnv *env, jobject obj, jint n, jint method, jint threshold);

@brief

Aplica el algoritmo indicado por method, con el valor de threshold indicado a la imagen número n, para calcular sus puntos de interés

@param

n: número de imagen

method: 0, 1 y 4 ->FAST

2 ->FastCV

3 ->GridORB

5 ->ORB compuesto

6 ->ORB

A. API del framework desarrollado

7 ->BRISK
8 ->GridBRISK

threshold: variación de tono a partir de la cual se considera punto de interés

jint findDescriptors (JNIEnv *env, jobject obj, jint n jint method);

@brief

Aplica el algoritmo indicado por method para la extracción de descriptores

@param

n: número de imagen

method: 0, 1 y 4 ->FAST

2 ->FastCV

3 ->GridORB

5 ->ORB compuesto

6 ->ORB

7 ->BRISK

8 ->GridBRISK

@return

número de descriptores indicados

textitjint findMatches (JNIEnv *env, jobject obj, jint method);

@brief

Aplica el algoritmo indicado por method para la búsqueda de emparejamientos

@param

n: número de imagen

method: 0 para *OpenCV FLANN* o 1 para *FastCV KDTree*

@return

número de emparejamientos encontrados

*jint homography (JNIEnv *env, jobject obj, jint method);*

@brief

Calcula una homografía

@param

method: método con el que se han calculado previamente los matches

@return

1 si se ha calculado

*jint checkHomography(JNIEnv *env, jobject obj, jint method);*

@brief

Muestra los emparejamientos que han votado por la homografía, es decir, que han sido considerados inliers

@param

method: método con el que se han calculado previamente los matches

@return

número de matches válidos para la homografía

Apéndice B

Prototipo final

A continuación se describe el funcionamiento del prototipo final de proyección de realidad aumentada sobre una superficie plana a través de diagramas de estados que explican toda la ejecución de la aplicación.

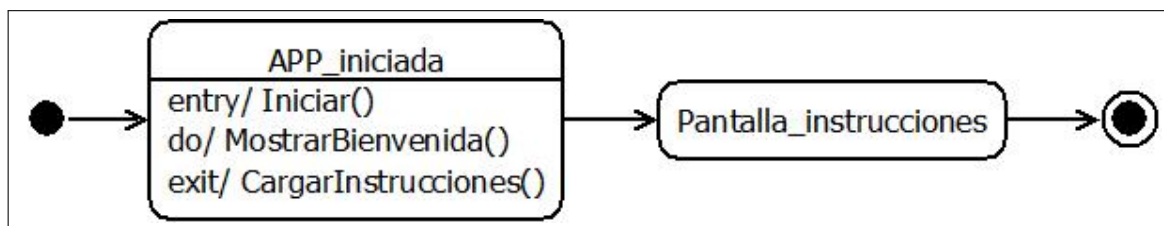


Figura B.1: El usuario inicia la aplicación y se le muestra una pantalla de carga y el funcionamiento.

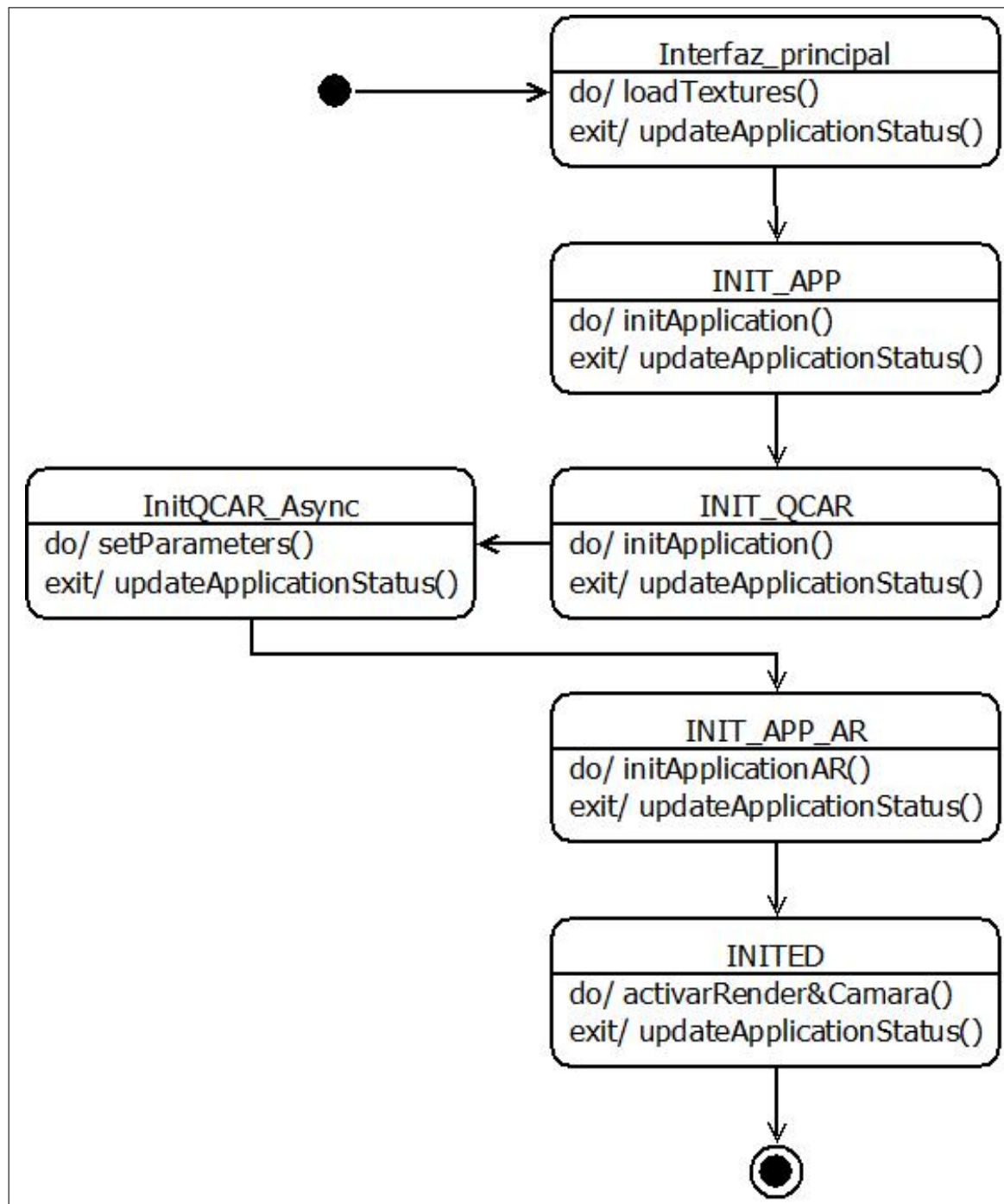


Figura B.2: Se cargan las texturas del modelo virtual y se inicializan todas las estructuras necesarias.

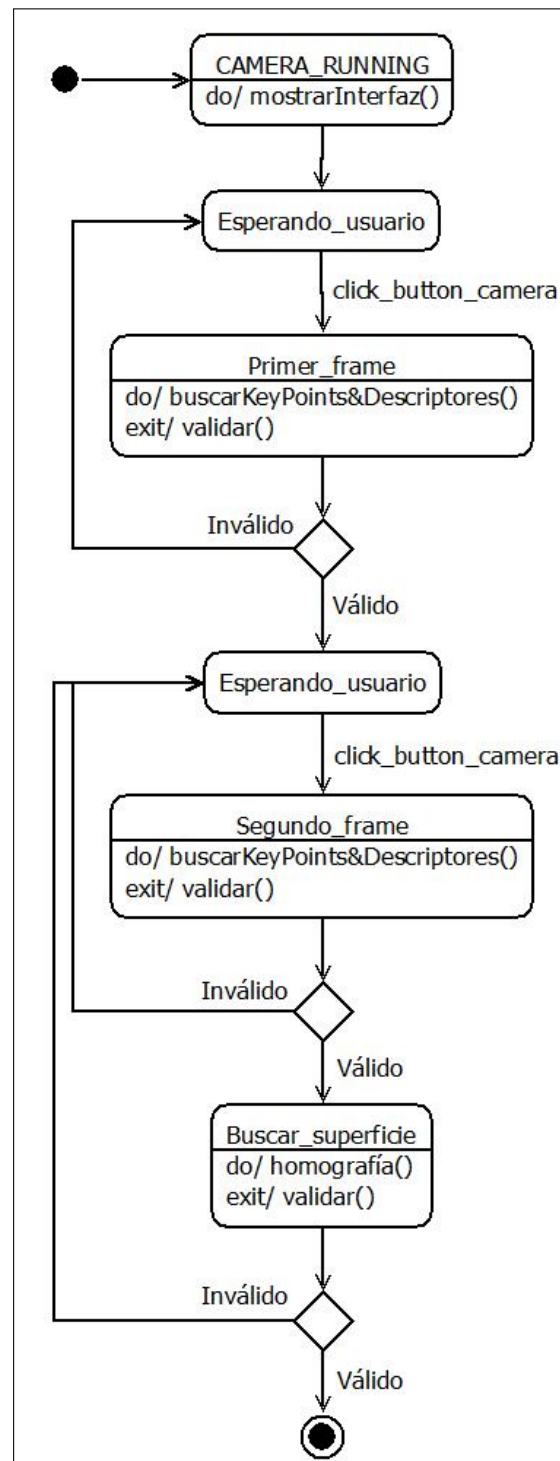


Figura B.3: Se muestra la interfaz principal, imagen 4.2. Al pulsar en la cámara se captura el frame actual y se obtienen sus puntos de interés y descriptores. Se hace lo mismo con un segundo frame y se trata de buscar una superficie plana en la escena.

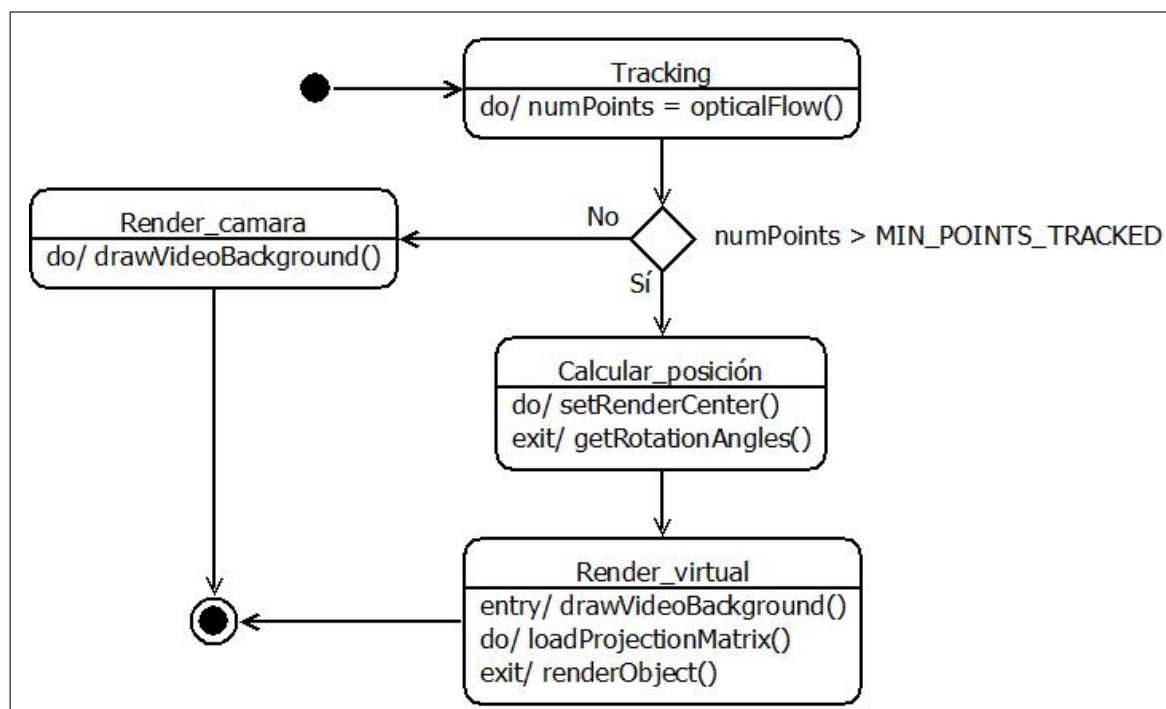


Figura B.4: Se trackea el plano en cada nuevo frame. Si se encuentra, se proyecta sobre él el modelo virtual.

Apéndice C

Pruebas del módulo de detección de una superficie plana

En este anexo se detallan las pruebas efectuadas para el módulo de detección de superficies planas. Dichas pruebas consisten en un banco de imágenes a las que se le han aplicado todos los métodos implementados bajo condiciones similares para comparar los resultados. Hay que destacar, sin embargo, que al haber sido realizadas en una máquina real, las medidas de tiempo pueden sufrir ligeras alteraciones entre las pruebas.

La estructura es la siguiente:

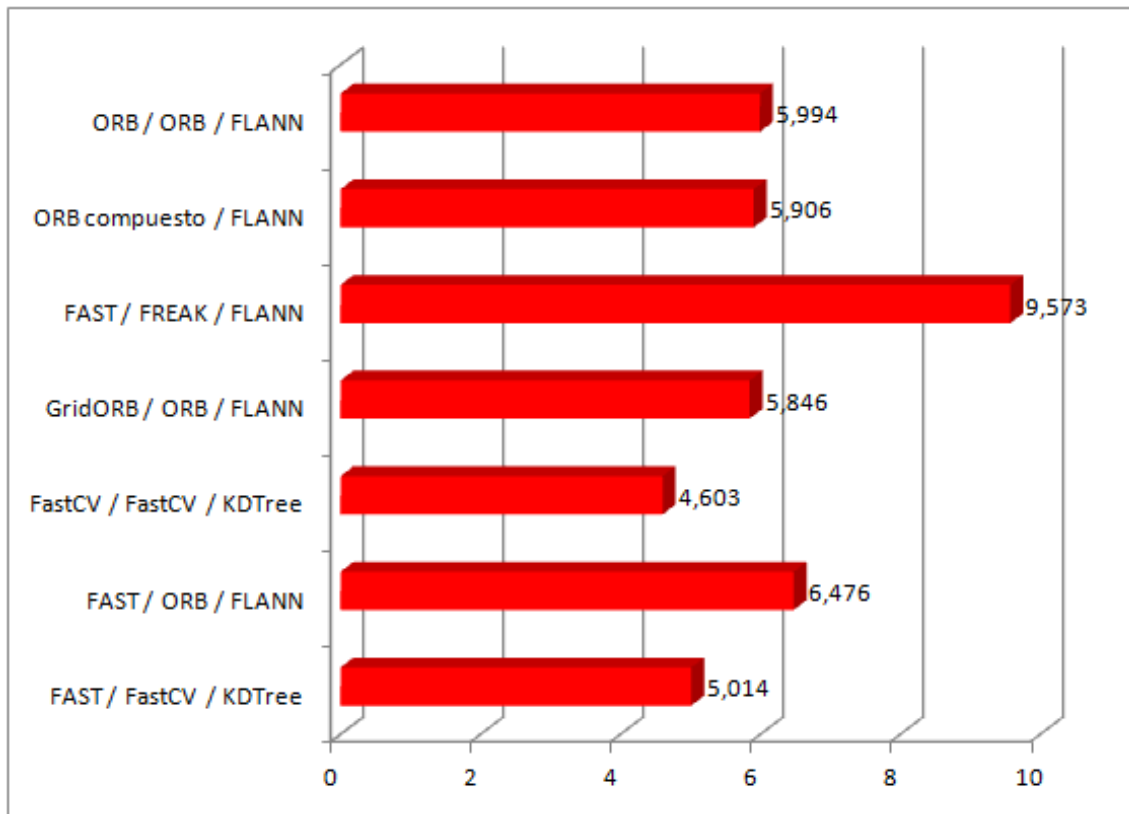
- Tabla resumen en la que se indica el nombre del escenario (que puede consultarse en la documentación digital en el directorio `/anexos/pruebas/[nombre]`). En ella se muestran los resultados obtenidos con cada método. Se muestran los puntos de interés obtenidos en cada frame, el tiempo de ejecución de cada uno, el tiempo de ejecución de los descriptores, el número total de emparejamientos realizados y su tiempo, y el número de emparejamientos válidos, es decir, los que han votado por la homografía, junto con el tiempo de cálculo de esta. Los tiempos son medidos en milisegundos. Todas las tablas pueden consultarse en el fichero incluido en la documentación digital en la ruta `/anexos/pruebas/prototipo1.xls`)

Se marcan en rojo los casos en los que el resultado no ha sido correcto, ya sea porque no se ha detectado ninguna superficie plana, o porque no se ha alcanzado el mínimo de 10 emparejamientos válidos. En naranja se muestran los casos en los que el resultado se ha dado por inválido por haber pocos emparejamientos, pero sin embargo estos son correctos.

- Gráfico con el tiempo total de ejecución del método, medido en segundos.
- Imágenes mostrando los emparejamientos totales y válidos obtenidos con cada método expresado como *puntos de interés / descriptores / emparejamientos*.

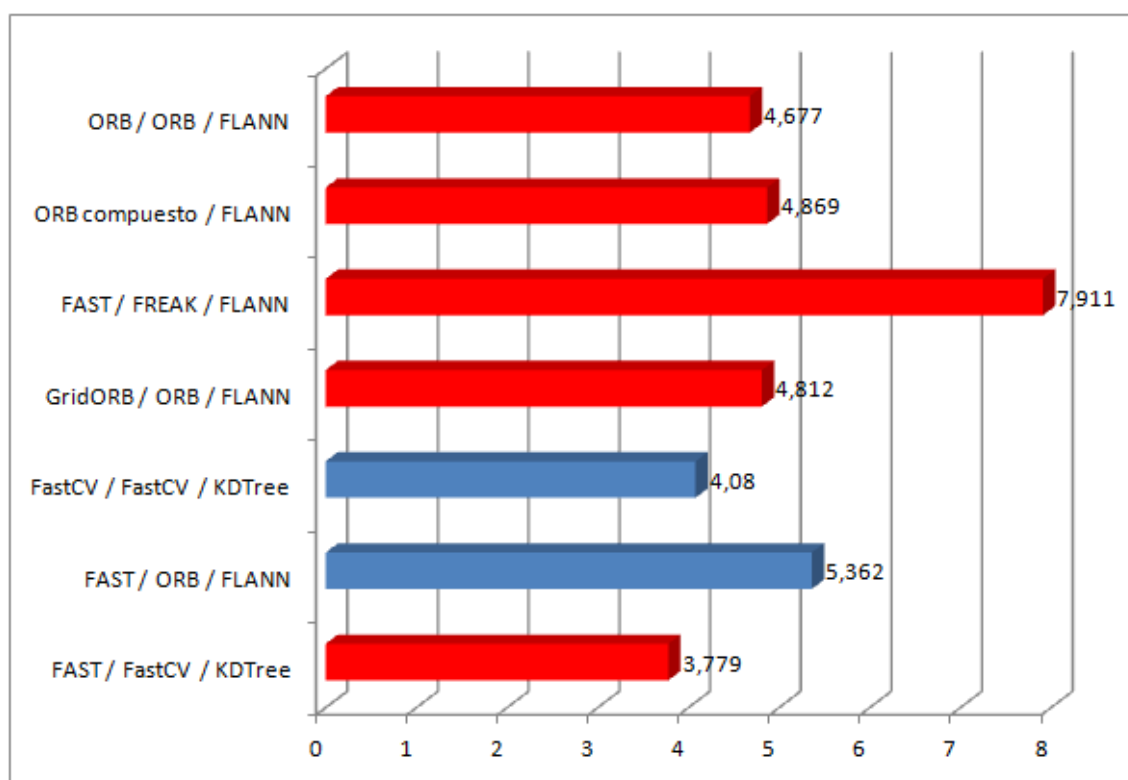
C. Pruebas módulo 1

Estatua Justicia	Puntos de interés				Descriptores	Emparejamientos				Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo	Tiempo	Matches	Tiempo	Válidos	Tiempo	
FAST / FastCV / KDTree	955	62	1000	45	73	407	81	7	4753	5014
FAST / ORB / FLANN	955	50	1000	41	613	485	796	102	4976	6476
FastCV / FastCV / KDTree	1000	51	1000	37	69	307	78	11	4368	4603
GridORB / ORB / FLANN	737	358	883	359	692	92	583	7	3854	5846
FAST / FREAK / FLANN	955	38	1000	43	2140	542	2176	53	5176	9573
ORB compuesto / FLANN	500	613	500	625		289	291	47	4377	5906
ORB / ORB / FLANN	500	299	500	371	607	289	287	48	4430	5994
BRISK / BRISK / FLANN	635	17802	734	16981	17941	195	800	16	4093	57617
GridBRISK / BRISK / FLANN	565	87886	635	8699	16418	72	590	5	3689	117282



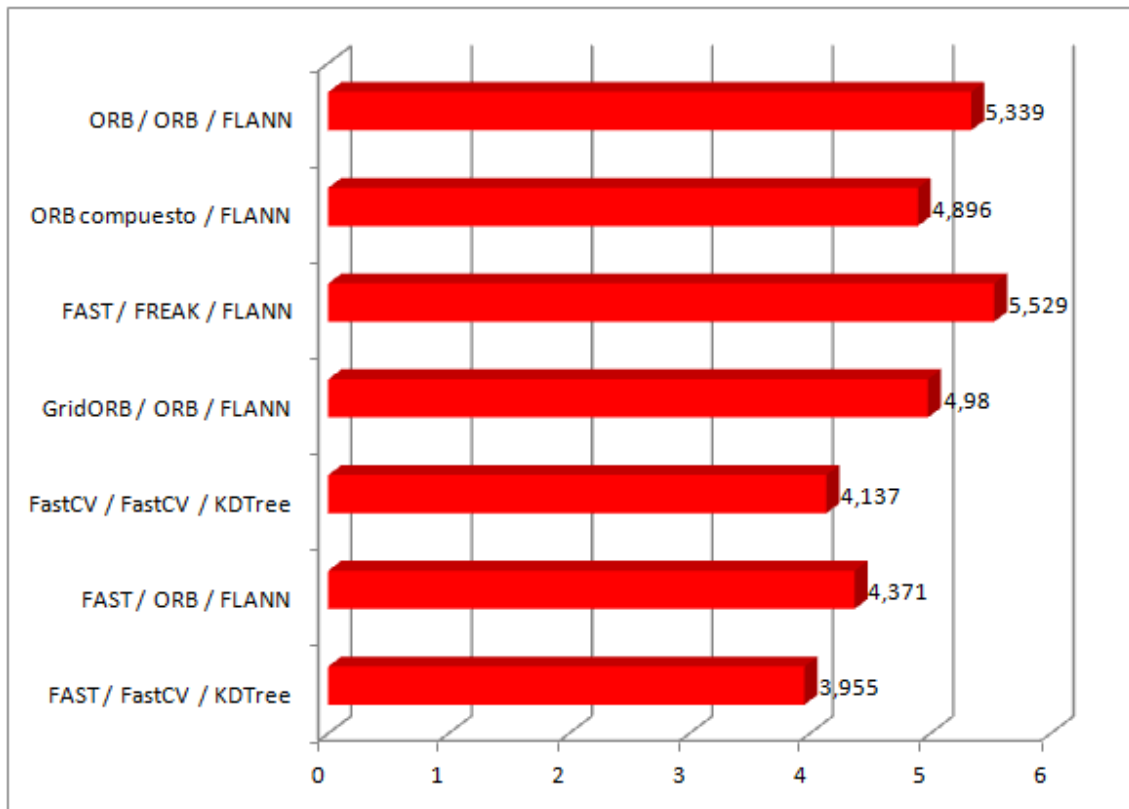
C. Pruebas módulo 1

Cartel Aragón	Puntos de interés				Descriptores	Emparejamientos		Homografía		Total	
	Imagen 1		Imagen 2			Tiempo	Matches	Tiempo	Válidos		Tiempo
	Puntos	Tiempo	Puntos	Tiempo							
FAST / FastCV / KDTree	766	43	784	40	58	39	57	6	3581	3779	
FAST / ORB / FLANN	766	42	784	39	462	277	459	12	4360	5362	
FastCV / FastCV / KDTree	1000	38	1000	27	64	134	74	15	3877	4080	
GridORB / ORB / FLANN	420	191	520	186	520	65	290	6	3625	4812	
FAST / FREAK / FLANN	766	37	784	39	2070	371	1193	6	4572	7911	
ORB compuesto / FLANN	500	444	500	457		69	291	8	3677	4869	
ORB / ORB / FLANN	500	186	500	200	633	69	293	8	3365	4677	
BRISK / BRISK / FLANN	390	18168	408	17396	18355	87	369	7	4034	58322	
GridBRISK / BRISK / FLANN	159	8869	221	8597	16802	25	126	5	3514	37908	



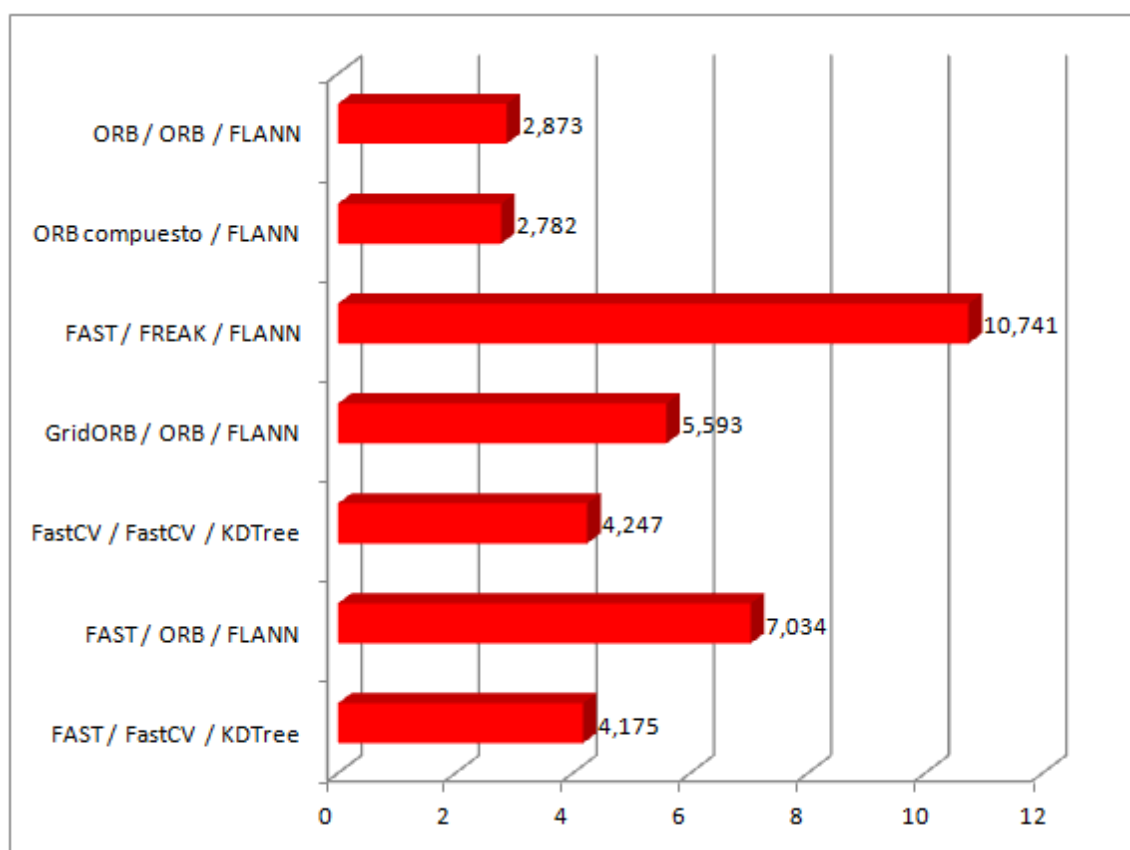
C. Pruebas módulo 1

Portal rejas	Puntos de interés				Descriptores	Emparejamientos				Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo						
FAST / FastCV / KDTree	295	36	416	38	47	71	41	11	3793	3955
FAST / ORB / FLANN	295	30	418	38	306	106	180	8	3817	4371
FastCV / FastCV / KDTree	822	30	1000	35	65	140	72	25	3935	4137
GridORB / ORB / FLANN	498	204	700	237	575	54	342	11	3622	4980
FAST / FREAK / FLANN	295	34	415	36	1088	150	419	6	3952	5529
ORB compuesto / FLANN	500	452	500	494		56	311	6	3639	4896
ORB / ORB / FLANN	500	195	500	233	610	56	463	6	3838	5339
BRISK / BRISK / FLANN	228	17462	278	17138	17502	37	171	6	3533	55806
GridBRISK / BRISK / FLANN	224	8585	274	8596	16711	21	198	5	3102	37192



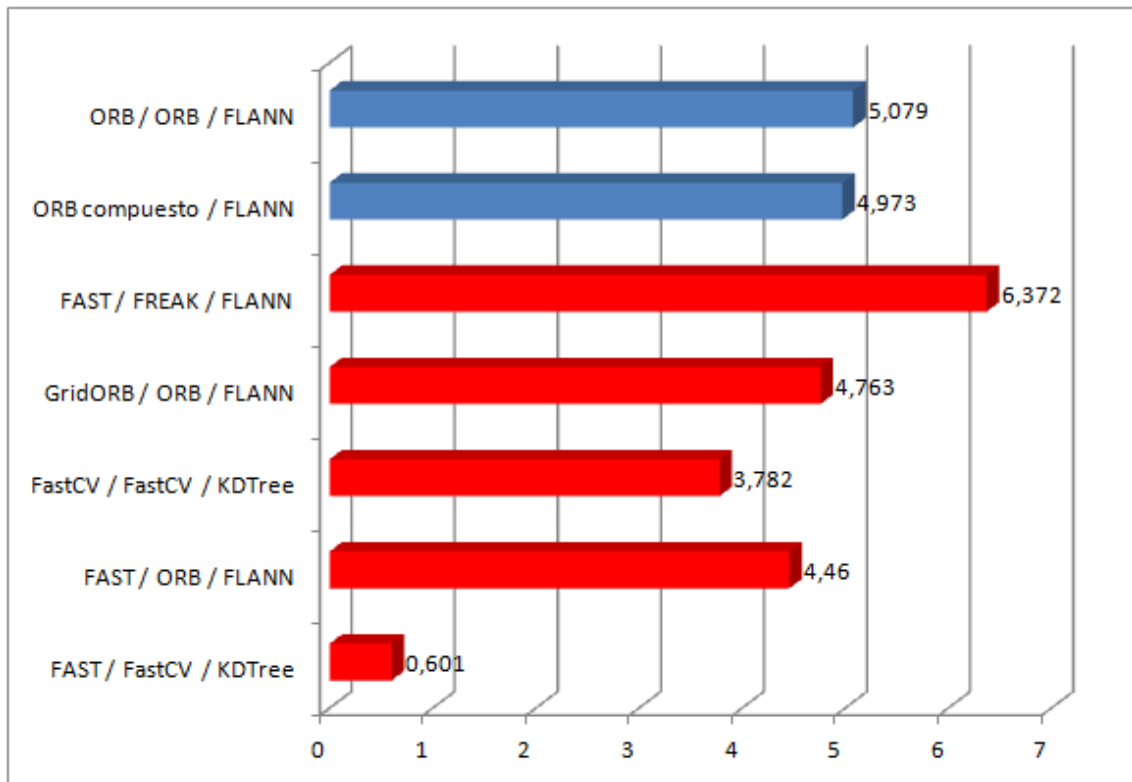
C. Pruebas módulo 1

Cartel Corte Inglés	Puntos de interés				Descriptores	Emparejamientos		Homografía		Total
	Imagen 1		Imagen 2			Matches	Tiempo	Válidos	Tiempo	
	Puntos	Tiempo	Puntos	Tiempo	Tiempo					
FAST / FastCV / KDTree	1000	39	1000	45	65	149	99	6	3927	4175
FAST / ORB / FLANN	1000	44	1000	45	682	517	1170	39	5093	7034
FastCV / FastCV / KDTree	1000	41	1000	28	65	158	75	11	4038	4247
GridORB / ORB / FLANN	788	324	748	354	655	73	561	15	3699	5593
FAST / FREAK / FLANN	1000	44	1000	45	2196	532	3302	19	5154	10741
ORB compuesto / FLANN	500	610	500	606		34	301	10	1265	2782
ORB / ORB / FLANN	500	329	500	362	604	34	286	10	1292	2873
BRISK / BRISK / FLANN	801	16975	816	16974	17133	108	1250	10	3861	56193
GridBRISK / BRISK / FLANN	665	8816	554	8717	16620	81	823	7	4073	39049



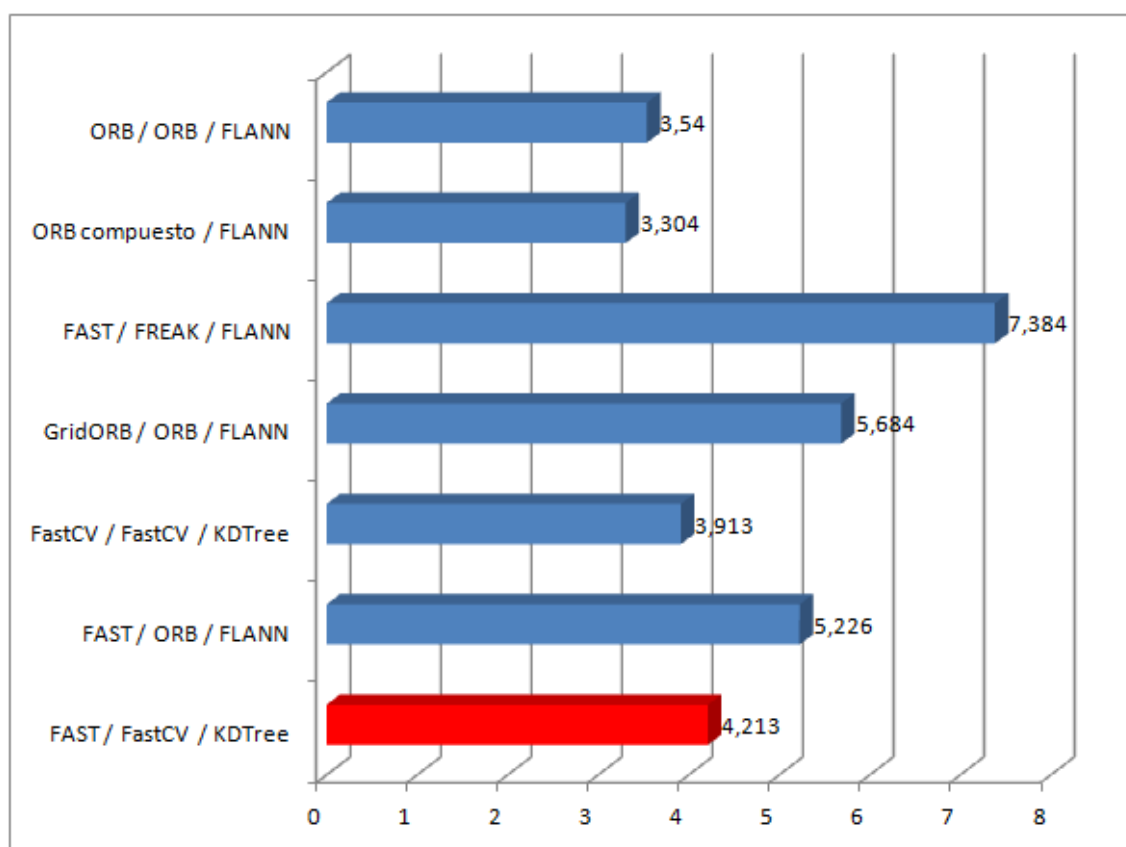
C. Pruebas módulo 1

Cartel Ibercaja	Puntos de interés				Descriptores	Emparejamientos				Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo						
					Tiempo	Matches	Tiempo	Válidos	Tiempo	
FAST / FastCV / KDTree	379	41	159	33	38	13	21	5	468	601
FAST / ORB / FLANN	379	43	159	41	272	174	120	7	3984	4460
FastCV / FastCV / KDTree	1000	56	580	34	61	63	70	11	3561	3782
GridORB / ORB / FLANN	328	209	205	173	464	51	120	9	3797	4763
FAST / FREAK / FLANN	379	57	159	54	2079	181	222	5	3960	6372
ORB compuesto / FLANN	500	412	500	408		131	326	25	3827	4973
ORB / ORB / FLANN	500	159	500	162	615	131	307	26	3836	5079
BRISK / BRISK / FLANN	236	16718	138	16436	16141	41	118	5	3950	53363
GridBRISK / BRISK / FLANN	135	8303	44	8565	17985	21	40	5	2846	37739



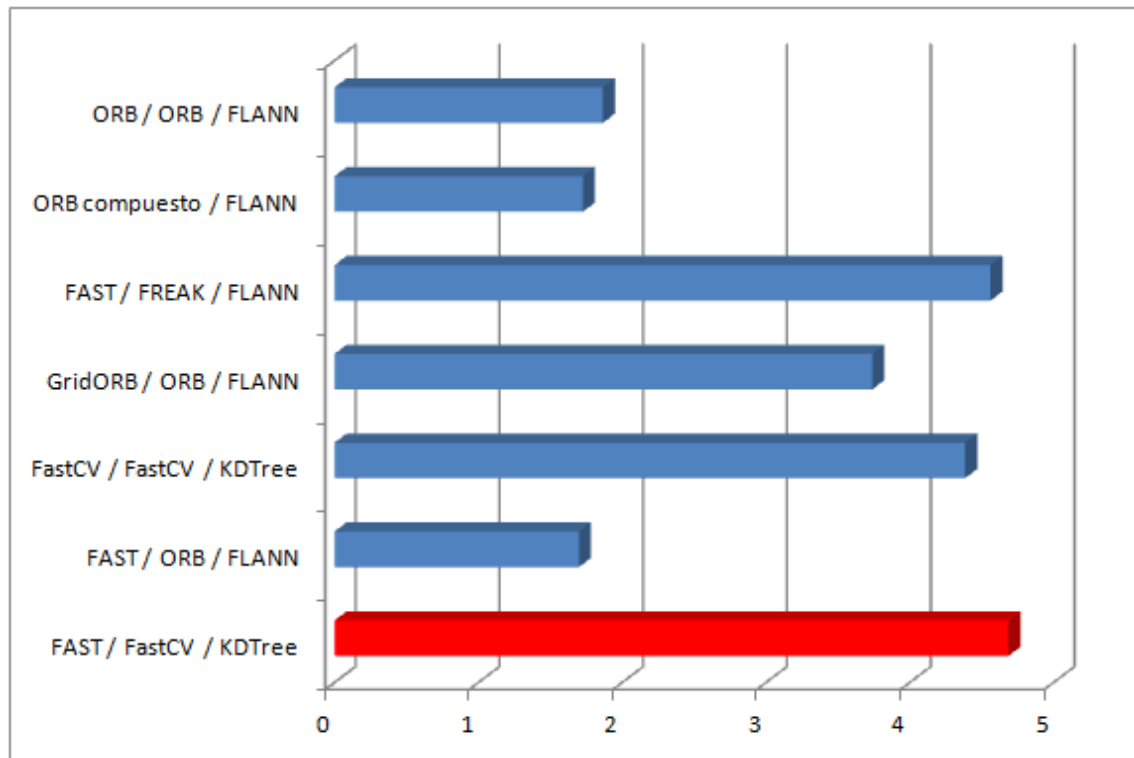
C. Pruebas módulo 1

Cartel 1€	Puntos de interés				Descriptores	Emparejamientos		Homografía		Total
	Imagen 1		Imagen 2			Matches	Tiempo	Válidos	Tiempo	
	Puntos	Tiempo	Puntos	Tiempo	Tiempo					
FAST / FastCV / KDTree	747	38	599	35	52	131	48	25	4040	4213
FAST / ORB / FLANN	747	40	599	36	473	272	381	16	4296	5226
FastCV / FastCV / KDTree	1000	43	1000	27	72	191	78	46	3693	3913
GridORB / ORB / FLANN	782	294	754	288	675	113	579	19	3848	5684
FAST / FREAK / FLANN	747	30	637	36	2047	327	825	36	4446	7384
ORB compuesto / FLANN	500	526	500	529		186	312	51	1937	3304
ORB / ORB / FLANN	500	271	500	265	619	186	298	50	2087	3540
BRISK / BRISK / FLANN	433	17069	354	18579	16852	66	330	9	3661	56491
GridBRISK / BRISK / FLANN	555	10258	463	8649	16969	88	576	6	3735	40187



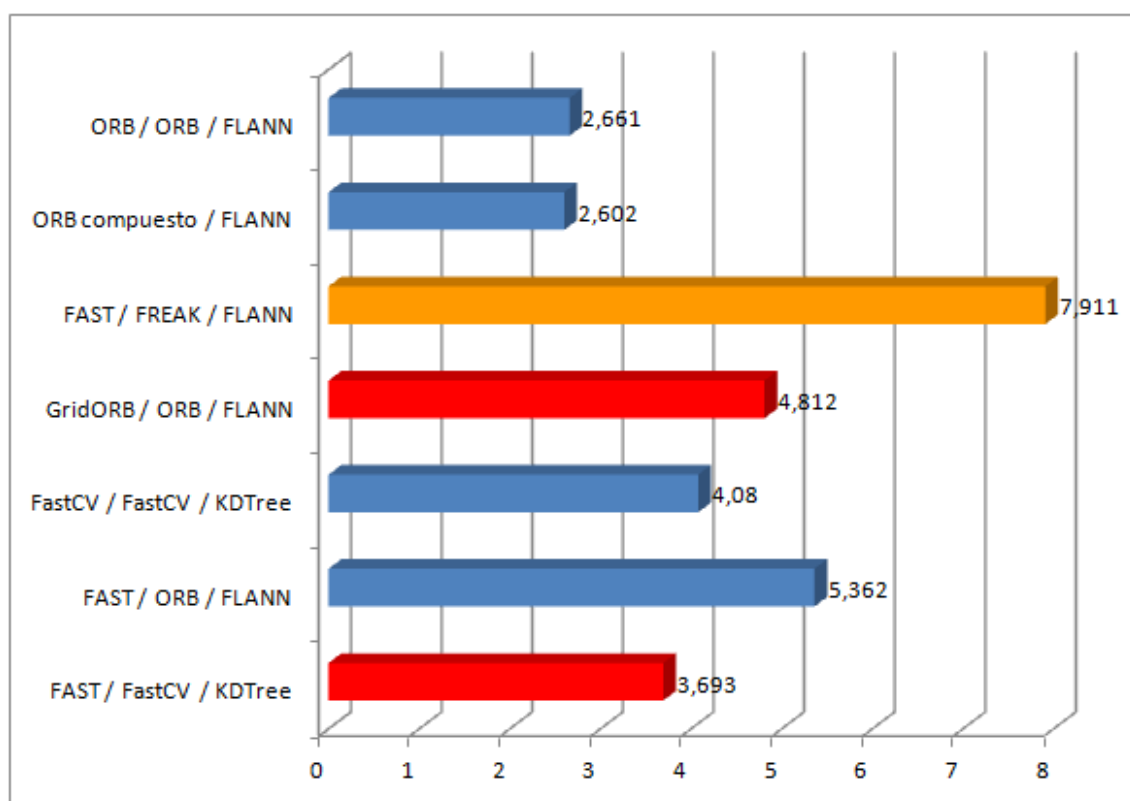
C. Pruebas módulo 1

Cartel pared	Puntos de interés				Descriptores	Emparejamientos				Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo	Tiempo	Matches	Tiempo	Válidos	Tiempo	
FAST / FastCV / KDTree	643	55	677	40	60	315	61	7	4471	4687
FAST / ORB / FLANN	643	67	677	42	543	476	481	219	567	1700
FastCV / FastCV / KDTree	1000	34	1000	36	76	562	87	43	4153	4386
GridORB / ORB / FLANN	631	251	688	260	658	125	434	33	2137	3740
FAST / FREAK / FLANN	643	43	277	45	2142	420	994	147	1339	4563
ORB compuesto / FLANN	500	470	500	465		267	317	107	476	1728
ORB / ORB / FLANN	500	207	500	259	614	267	305	107	482	1867
BRISK / BRISK / FLANN	407	16742	377	16970	17857	135	345	32	3343	55257
GridBRISK / BRISK / FLANN	321	8483	305	8375	16183	71	255	12	3672	36968



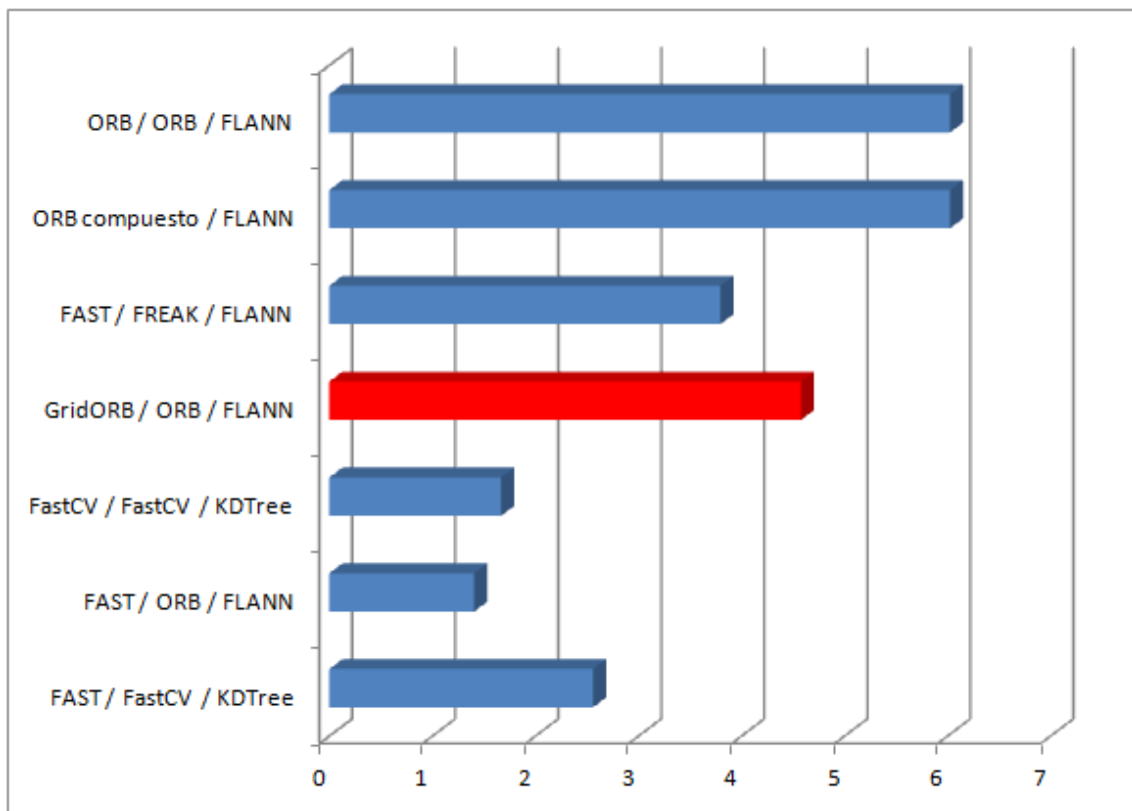
C. Pruebas módulo 1

Cartel Citybank	Puntos de interés				Descriptores	Emparejamientos		Homografía		Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo		Tiempo	Matches	Tiempo	Válidos	
FAST / FastCV / KDTree	105	29	103	32	38	32	7	5	3587	3693
FAST / ORB / FLANN	766	42	784	39	462	277	459	12	4360	5362
FastCV / FastCV / KDTree	1000	38	1000	27	64	134	74	15	3877	4080
GridORB / ORB / FLANN	420	191	520	186	520	65	290	6	3625	4812
FAST / FREAK / FLANN	766	37	784	39	2070	371	1193	6	4572	7911
ORB compuesto / FLANN	484	385	479	385		360	317	90	1515	2602
ORB / ORB / FLANN	484	157	458	146	606	304	281	91	1471	2661
BRISK / BRISK / FLANN	103	16689	106	16445	16353	34	58	7	3595	53140
GridBRISK / BRISK / FLANN	159	8869	221	8597	16802	25	126	5	3514	37908



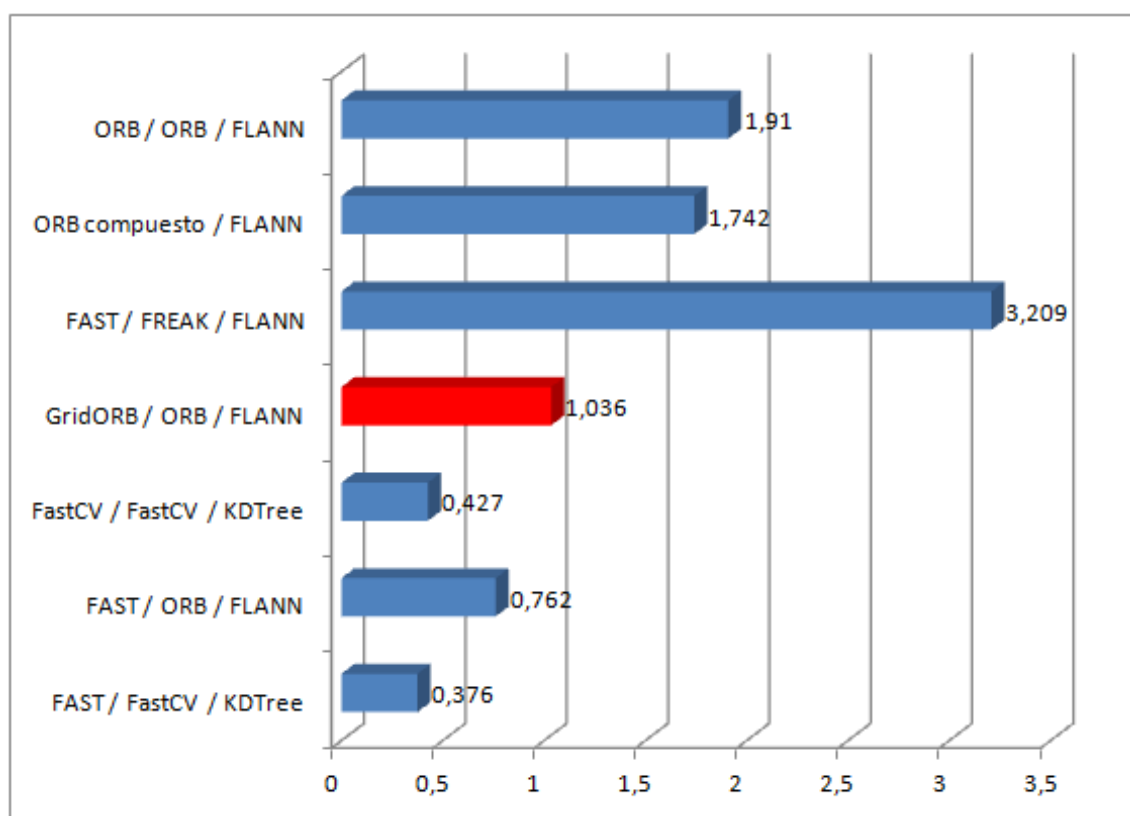
C. Pruebas módulo 1

Cartel Niketos	Puntos de interés				Descriptores	Emparejamientos				Total
	Imagen 1		Imagen 2							
	Puntos	Tiempo	Puntos	Tiempo	Tiempo	Matches	Tiempo	Válidos	Tiempo	
FAST / FastCV / KDTree	137	30	108	32	33	63	10	16	2455	2560
FAST / ORB / FLANN	137	36	112	32	197	93	47	29	1094	1406
FastCV / FastCV / KDTree	448	40	311	26	50	214	38	63	1514	1668
GridORB / ORB / FLANN	217	184	203	176	412	86	110	7	3696	4578
FAST / FREAK / FLANN	137	39	112	34	2038	95	102	17	1583	3796
ORB compuesto / FLANN	500	514	500	418		405	338	63	4755	6025
ORB / ORB / FLANN	500	172	500	176	607	405	340	64	4731	6026
BRISK / BRISK / FLANN	130	17703	113	18204	16780	93	122	15	4100	56909
GridBRISK / BRISK / FLANN	63	8474	41	8406	16409	23	36	4	3545	36870



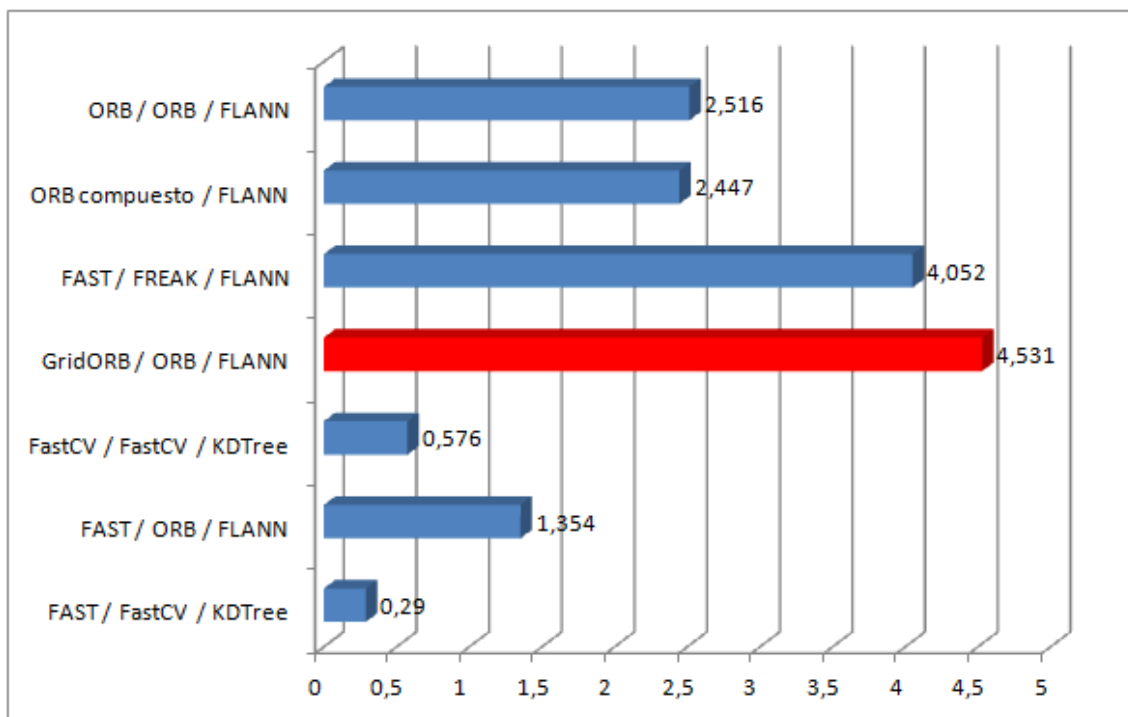
C. Pruebas módulo 1

Plaza Paraíso	Puntos de interés				Descriptores	Emparejamientos		Homografía		Total	
	Imagen 1		Imagen 2			Tiempo	Matches	Tiempo	Válidos		Tiempo
	Puntos	Tiempo	Puntos	Tiempo							
FAST / FastCV / KDTree	262	43	117	34	35	46	24	22	240	376	
FAST / ORB / FLANN	262	34	117	34	238	89	65	37	391	762	
FastCV / FastCV / KDTree	730	35	362	39	57	198	38	96	258	427	
GridORB / ORB / FLANN	416	166	294	168	465	6	192	4	45	1036	
FAST / FREAK / FLANN	262	40	117	38	2068	116	131	38	932	3209	
ORB compuesto / FLANN	500	414	500	397		132	282	53	649	1742	
ORB / ORB / FLANN	500	165	500	159	611	132	307	53	668	1910	
BRISK / BRISK / FLANN	158	17105	65	17948	16637	21	64	6	1484	53238	
GridBRISK / BRISK / FLANN	234	8532	66	8584	16432	9	74	4	273	33895	



C. Pruebas módulo 1

Tarjeta	Puntos de interés				Descriptores	Emparejamientos									
	Imagen 1		Imagen 2								Tiempo	Matches	Tiempo	Homografía	
	Puntos	Tiempo	Puntos	Tiempo										Válidos	Tiempo
FAST / FastCV / KDTree	183	32	168	41	36	55	14	29	167	290					
FAST / ORB / FLANN	183	42	168	39	222	141	83	46	968	1354					
FastCV / FastCV / KDTree	861	24	781	26	63	493	57	233	406	576					
GridORB / ORB / FLANN	231	168	245	165	403	83	36	12	3759	4531					
FAST / FREAK / FLANN	183	43	168	33	2163	134	156	38	1657	4052					
ORB compuesto / FLANN	500	423	500	413		345	327	108	1284	2447					
ORB / ORB / FLANN	500	151	500	165	613	345	327	108	1260	2516					
BRISK / BRISK / FLANN	119	17149	102	19069	16912	17	55	5	1299	54484					
GridBRISK / BRISK / FLANN	41	8421	41	8517	17021	0	-	-	-	-					



C. Pruebas módulo 1





Apéndice D

Pruebas del módulo de realidad aumentada

En este anexo se detallan los resultados obtenidos en los distintos escenarios en los que se ha testado el prototipo 2. Se muestran después capturas de pantalla del módulo de proyección de realidad aumentada. En ellas se ven proyecciones del modelo virtual sobre distintas superficies, obtenidas durante la ejecución de las pruebas del prototipo 2.

Input			Output	
Nombre	Dificultad	Descripción	Superficie	Orientación
Nombre de secuencia:		Detalles relevantes sobre la secuencia	Válido si el modelo virtual se muestra sobre la superficie con un tamaño adecuado	Válido si el modelo virtual se muestra con orientación correcta, es decir, paralelo al plano y apoyado en él
el nombre se corresponde con la carpeta donde están las imágenes del test	1 - Fácil			
	2 - Medio			
	3 - Difícil			
tarjeta_qbitera_+_tarjeta_zgz_activa	2	* Dos posibles superficies con textura * Escena sin textura	Válido	Válido
terrazza_servilletero	3	* Plano con suficiente textura * Escena con mucha textura	Inválido	Válido
			El modelo virtual aparece en cualquier lugar de la escena.	
			Homografía y matches mal calculados	
Abogados	2	* Plano con suficiente textura, pero confusa * Escena sin textura	Válido	Válido
Asterix_Obelix	1	* Plano con suficiente textura * Escena sin textura	Válido	Válido
Cartel_1euro	2	* Plano con suficiente textura * Árboles en la escena	Válido	Válido
Cartel_100montaditos	3	* Plano con suficiente textura, pero confusa * Árboles en la escena	Válido	Válido
Cartel_verano	3	* Plano con poca textura y muy confusa. * Patrones repetidos en el plano	Válido	Válido
Citi	2	* Plano con suficiente textura * Plano borroso tras un cristal	Válido	Válido
Decathlon_terrazza	3	* Plano no dominante en la escena	Inválido	Válido
			El modelo virtual aparece en cualquier lugar de la escena.	
Edificio_caja_castilla_la_mancha	3	* Plano con textura muy confusa. Patrones muy repet. * Árboles en la escena	Válido	Válido
		* Varios posibles planos dominantes		
Edificio_coso	3	* Plano con textura confusa. Patrones repetidos.	Válido	Válido

D. Pruebas módulo 2

El_Pilar	3	* Plano con textura confusa. Patrones repetidos. * No hay ningún plano dominante en la escena.	Válido	Válido
Farmacia_Barrau	2	* Plano con poca textura y muy confusa. * Escena sin textura (se ha evitado la persiana en la segunda imagen)	Inválido	-
Folleto_hoy_salgo	1	* Plano con suficiente textura, aunque confusa * Escena con mucha textura, pero fácil de filtrar al existir un claro plano dominante a menor distancia	Válido	Válido
Folleto_informacion_turistica	1	* Plano con suficiente textura, aunque confusa * Escena con mucha textura, pero fácil de filtrar al existir un claro plano dominante a menor distancia	Válido	Válido
Foster	2	* Plano con suficiente textura, aunque confusa * Escena con mucha textura, pero fácil de filtrar al existir un claro plano dominante a menor distancia	Válido	Válido
Heraldo	2	* Plano con escasa textura, y confusa * Escena con textura, a diferente distancia y en varios planos no dominantes	Válido	Válido
Ibercaja_arbol	3	* Plano con escasa textura, y muy confuso. Patrón altamente repetido. * Árboles en la escena	Inválido	-
Kuhnel	2	* Plano con suficiente textura * Escena con mucha textura, y en el mismo plano	Válido	Válido
Lord_of_the_Rings	1	* Plano con suficiente textura * Escena sin textura	Válido	Inválido
Mapa_fail	3	* Plano con suficiente textura, pero extremadamente confusa. Se repiten patrones por toda la superficie	Inválido	-
Menu_espumosos	2	* Plano con suficiente textura, pero con patrones muy repetibles * El plano dominante objetivo está dentro de otro plano igualmente dominante	Válido	Válido
Muñeca_rara	3	* Plano con mucha textura, pero con patrones muy repetibles * Aparece exactamente el mismo plano dos veces en la misma imagen	Válido	Válido
Niketos	1	* Plano con mucha textura * Escena sin textura	Válido	Válido
Parking	3	* Plano con suficiente textura * Escena con mucha textura * Plano objetivo sobre otro plano dominante	Válido	Válido
PV_desigual	3	* Plano con suficiente textura * Árboles en la escena * Varios posibles planos	Válido	Válido
Rebajas_corte_ingles	3	* Plano pequeño y alejado, insuficiente textura * Árboles en la escena	Inválido	-
Rebajas_corte_ingles_bueno	2	* Plano con suficiente textura * Escena con textura en el mismo plano	Válido	Inválido

D. Pruebas módulo 2

Santa_Engracia	3	* No hay ningún plano dominante en la escena. * Monumento: se considera toda la fachada como superficie donde poder colocar el modelo virtual	Válido	Válido
			Aunque el modelo se muestra sobre una torre, sí aparece dentro de la fachada, lo que se considera válido en el caso de una escena de un monumento.	
Torre_ayuntamiento	3	* No hay ningún plano dominante en la escena. * Monumento: se considera toda la fachada como superficie donde poder colocar el modelo virtual	Válido	Válido
			Aunque el modelo se muestra sobre una torre, sí aparece dentro de la fachada, lo que se considera válido en el caso de una escena de un monumento.	
Zepelin	2	* Plano con suficiente textura * Demasiada textura en la escena	Inválido	Válido
			Aunque el modelo se muestra, y en la imagen de ejemplo tomada aparece incluso dentro del plano, en un caso general no se garantiza que se muestre dentro.	

* Los resultados marcados en color naranja indican que, a pesar de proyectarse el modelo virtual de forma correcta en la pantalla, no se ha calculado bien la superficie. Se muestra correctamete solo debido a las propiedades especiales de la escena.

